2000

Towards a kernel-architecture for modelling systems.

Paul David. Preney
University of Windsor

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

TOWARDS A KERNEL-ARCHITECTURE FOR MODELLING SYSTEMS

by

Paul David Preney

A Thesis
Submitted to the College of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario
Canada
1999

Abstract

A kernel has been developed to enable the construction of distributed and parallel software for sequential, distributed, and parallel (virtual) machines. Additionally, a simple graphical user interface is provided to demonstrate two example programs written using the developed kernel. The example programs, although simple, sufficiently demonstrate that the resulting kernel design can be used to construct and deploy a complete distributed and parallel (virtual) machine. The kernel as developed in this thesis serves as a proof-of-concept prototype based on an underlying abstract design.

Dedication

To my parents,
To my teachers,
To my friends

To all those who cared to give
The time,
The patience,
The effort,
And the love

To hear,
To speak,
And to understand

The unheard
And oft forgotten

It is to you that this
Thesis is dedicated

Acknowledgements

Mom and Dad, I thank you for never giving up; for your love, courage, determination, and support; and for the graciously given gifts needed to hear, speak and understand the unheard. Without you both, I, most certainly, would have never had even the opportunity to enter any post-secondary academic institution. Dr. Kent, Dr. Tsin, Dr. Tjandra, and Dr. Baylis, I thank you for your comments, questions, and criticisms of this thesis. I would also like to thank Mr. Doug Thistle for providing "another set of eyes" to proofread this thesis.

Table of Contents

List of Figures

Document Conventions Used in this Thesis

It was necessary to adopt some writing conventions in this thesis to remain terse and succinctly communicate with the reader. The constant-width typeface `Letter Gothic` is used for source code, class hierarchy definitions, class and method names, or, for references to terms in a specific computer programming language or pseudocode. Usually such text is indented:

```
public interface AnExampleInterface
{
    Object getProperty();
    void setProperty(Object anObject);
    boolean isNull();
}
```

However at times it is necessary to use source code, class and method names, references to programming language terms, and/or pseudocode within the thesis body text proper. When such is employed, these items always appear in the `Letter Gothic` typeface. For e.g., "An `IteratorForEmptyContainer` represents a zero-length sequence." The term `IteratorForEmptyContainer` is a Java class that is defined in this thesis (see page 74 for the complete text). All of the remaining text is presented in the proportionately spaced, serif typeface of Classical Garamond having standard English-language semantics.

CHAPTER 1.
Introduction

1.1. Purpose of the Thesis

The focus of this thesis aims to partially solve the following problem: "How can one design and implement a *computing system* that minimizes the work in designing, implementing, and maintaining *any* computing system *including* itself?" Such a system, if indeed one exists, would be of enormous benefit to not just computer scientists, but, also to society as a whole, for computing systems are ubiquitous in society. For example, corporations are systems that compute, i.e., manufacture and produce products and services; personal computers compute the execution of various softwares; schools compute the dissemination of knowledge and skills; etc. Essentially, a computing system is a group of interacting entities that perform some specific task and therefore may require some set of inputs and may produce some set of outputs.

From the perspective of a philosopher or scientist, a computing system can be said to be a machine that performs some set of tasks requiring some kinds of input which produces some types of output. Each task merely computes something, which includes "nothing", by some unspecified means. If the task is being performed by a computer, then it is likely a function, procedure, or a relational operator. If the task is being performed by a human, then it could also be theorem proving, modelling, walking, enjoying music, etc. as appropriate in a given situation. Each task found within a computing system, may interact, forming a network with other tasks in order to "compute" what the computing system must. Since a computing system is a network (of tasks), it therefore has topological structure. Further, computing systems may be considered "tasks" themselves; i.e., a computing system may interact with other computing

systems.

It is important to note that to consciously *use* a computing system it has to be recognized first so as to permit its definition. Once it is defined, its properties and behaviours can be studied, e.g., ideally with the scientific method, in order to better understand that computing system. Then, in order to *actually use* a defined computing system, it must somehow exist; —either by creating an implementation of it if possible, or that it exists already. Interfaces must be provided to properly communicate all necessary "input" and "output" to the computing system, otherwise the system may not be of any use. Obviously, the *encoding* of such inputs and outputs is vital to appropriately manipulate that computing system. If the input encoding is not understood, then at best garbage output is returned. If, on the other hand, the output is not understood, then the reader of such will not be able to understand its meaning without analysis and insight.

The goal of designing and implementing a complete computing system as discussed above however is beyond the scope of this thesis. It is not even known whether or not it is possible to state at this time that such a system even exists. Instead, keeping the aforementioned goal in mind, the objective of this thesis is to design an architecture that should be reflected and used in such a computing system's kernel. Additionally, the computing system's kernel in this thesis should support the minimization of work "in designing, implementing, and maintaining *any* computing system *including* itself." This means that the kernel *must* allow for the possibility to model and implement, i.e., encode, any computing system including itself. Restated, the design of the kernel presented in this thesis, must allow for the modelling of arbitrary computing systems. Since this includes modelling itself, the kernel design presented in this thesis is a hypothetical basis for any *metamodelling* (computing) system.

1.2. Justification of This Thesis' Purpose

It may not be immediately obvious why one would even assert a need to research and create a system towards minimizing the work involved to design, implement, and maintain an arbitrary computing system. To create or design something is to construct a model and possibly, an implementation for such. The tasks of creating and designing happen so often that one usually doesn't even think about it; —this is especially true when using computers.

The utility of a computer is driven by one's ability to write software to fulfill such purposes. It makes sense that the easier it is to facilitate the design, implementation, and maintenance of software, the easier it will be to design, implement, and maintain software. Similar reasoning holds true for computing systems which can contain computers. This is the single most important practical goal of pursuing this line of research.

Since it is important how input to a computing system is encoded and how its output is decoded, it is of equal importance to be able to *design* the appropriate computing system input and output *interfaces* from/to other interacting computing systems. At the very least, in computing systems involving human beings and computers, appropriate human-computer interfaces must be present so that human beings and computers can easily communicate through a variety of protocols with one another. Of course, there are machine-to-machine interfaces as well. Concerning oneself with the design of *interfaces* to computing systems, forces the researcher to acknowledge the design(s) of how those interfaces *interface* with each other. Specifically, designing *interfaces* is called *modelling* and designing how *interfaces interface* is called *metamodelling*. As this thesis is concerned with the discovery and implementation of ideal metamodelling systems, modelling and metamodelling issues are

paramount. Hence, this is also an important justification for pursuing this line of research.

There are a number of other relevant justifications as well. These include: (i) to create a system that is "self-aware", not necessarily to create a "thinking" being, but, rather to automate and maintain new implementations of pre-existing designs; (ii) to create a system that facilitates the automation of information addition, retrieval, and purging to/from data stores; (iii) to create a system that can track the evolution of both model designs and implementations; and (iv) to create a system that can, as much as possible, automate how software is deployed across and within computing systems dynamically from a sufficient specification (i.e., sequential, parallel, distributed, etc.).

Obviously, it is an enormous undertaking to even attempt to accomplish the goals mentioned above as well as those that have not been mentioned. To maximize the chance of realizing these goals, this thesis narrows its focus to begin designing and implementing a computing system architecture and specific kernel operations that will serve to facilitate metamodelling.

CHAPTER 2.
Fundamental Concepts

The terms *model* and *modelling* mean different things to different persons. This is easily verified by scanning the literature looking for the keywords *model* and *modelling*; —one obtains nearly everything that has been written regarding computer graphics, programming, network, and software design, etc. In essence, everything ever created through the guise of engineering or science has been based on a model, and thus, the specification thereof is the process of modelling. This essential fact unfortunately complicates the task of generating a formal literature survey on the relevant topics as it pertains to either this thesis or its general research topic. This chapter serves as an overview of the literature and also provides the necessary foundation to understand and appreciate fully the rest of this thesis document. Please note that security and job scheduling are beyond the scope of this document.

2.1. Definition of Model, Modelling, Implementation, & User

For the computing sciences, perhaps the most suitable definition of what a model is, both abstractly and existentially, is given by Date and Darwen:

> A **model** is an abstract, self-contained, *logical* definition of the objects, operators, and so forth, that together constitute the abstract machine with which users interact. *Note:* The term "objects" here is generic —it is not meant in its object-oriented sense.
>
> An **implementation** of a given model is the *physical* realization on a real computer system of the components of that model.
>
> — [DATE98]

Additionally, modelling is defined as, "to make a model (representation) of [something]" [LEXIC87]. Noting that "virtual" means "abstract" and that a "computing system" is a "machine", then a model defines a *virtual computing system*. Furthermore, a model's

existential form is its implementation, i.e., its *encoding* or "physical realization" of its abstract definition. Consider the following strings:

ten
1111111111
1010
10
A

These strings represent five separate encodings (i.e., implementations) of the logical definition (i.e., model) of the number "ten"; —in English, base 1 (unary), base 2 (binary), base 10 (decimal), and base 16 (hexadecimal) respectively. One should note that while thinking abstractly about a model there is no need to worry about *how* the model is implemented since one is only concerned with its definition; e.g., if one is aware of what the number "ten" is. However, when required to perform an operation with the number "ten", "ten" must be encoded somehow so that the computing system that performs such (e.g., "multiply input by 2") can be defined *and* implemented to operate with that encoding. This *does* have a direct effect on the amount of time, space, and energy required to perform that operation in a given computing system. It is imperative to define models independent of any of their implementations, else one has merely defined an implementation. Such clarity of definitions is of paramount importance if one wishes to implement a well-defined, consistent computing system that is both transparently object-oriented and relational [DATE98].

A computing system cannot be of practical utility unless there is an entity that exists outside of that computing system and interacts with it. Such an entity is called a *user* and usually refers to a human being although this need not be the case. A user interacts with a computing system by providing properly encoded input, interpreting any generated output, and by

providing a proper operating environment for that system.

2.1.1. Examples of Modelling in the Literature

Many areas of modelling research and development focus on data visualization. A data visualization machine is one that accepts some data and allows the human user to "see", "hear", "touch", "smell", or "taste" the data in order to better interpret the data set [KALAW93]. In fact, any implementation that produces output is considered a visualization machine provided that some user *interprets* such output through an appropriate input medium [KALAW93]. For practical and technological reasons, usually such visualizations are limited to the visual or auditory senses when the user is a human being. Much of the visual data is presented as either two- (e.g., [ONODE90]) or three-dimensional (e.g., [SCHRO96, SCHRO98]) images, possibly animated. When multiple visualizations of such data sets are possible, one can explore a data set from different perspectives, thus, expanding the ability for the user to better understand and present to others that data set [SCHRO98]. Thus, experimental scientists are better able to discover, design, and evaluate (e.g., simulate) theoretical models [ALMST96, HYDE96], teachers are better equipped to impart knowledge and understanding to students [HYDE96, NAPS96], and others are even able to communicate their data to interested parties through a medium such as the World Wide Web [HALL96, MATHE96a, MATHE96b, SCHAT97, WOOD96] via technologies like VRML [AMES97]. In essence, modelling either *enables* or *is* a process of *discovering* and *designing* definitions (i.e., models). The well known scientific method [CAMPB90, RAVEN89] is a modelling process.

Models, implementations, and the modelling process can be found across a wide variety of subject areas in the literature. In addition to some of the examples mentioned in the previous

paragraph, some of these subject areas include: search engine and database query (e.g., [GRAEF93, GRAEF94, GRAEF96, SCHAT97]), population biology and ecosystems (e.g., [FORGA96, LEVIN97]), medical (e.g., [CHI96, WEINS97]), weather (e.g., [MAX95, WITTE96]), computer circuit topology (e.g., [AGARW99]), multibody systems (e.g., [EBERH96]), tools for hypermedia (i.e., dynamic exploration) (e.g., [BOSSA96, LI96]), international phone calling fraud (e.g., [EICK96]), geographical & soil (e.g., [FAUST96, LI93]), circuit design and visualization (e.g., [ARSIN96]), adaptive human interface design (e.g., [ENCAR95]), computer programming (e.g., [SHANB97, TOPCU98a]), and system specification, metamodelling, and virtual machine research (e.g., [BRONT95, CHEN76, CREAS96, FORMA94, FORMA99]). Additional examples are easily found in science and engineering literature.

2.2. The Turing Machine Model & Its Limitations

In 1935 a computing model was proposed by Alan Turing. This abstract machine is one that has unlimited and unrestricted memory and is fully capable of accomplishing anything that modern, digital computers are capable of as all of those are, in essence, Turing machines [COPEL99, SIPSE97]. Alan Turing's purpose in proposing such a machine was to have a simple model fully capable of computing any calculation that a mathematician could; —provided that such was performed by some specific algorithm with limited time and energy in an unintelligent, yet disciplined, manner [COPEL99]. Since Turing's model describes all such Turing machine implementations, his model is now known as the *universal Turing machine.*

2.2.1. The Turing Machine Model

All Turing machines consist of a processor with a "tape head" that can read and write

symbols as well as move an infinitely long tape. Prior to starting a computation, the tape contains only an input string and is otherwise marked blank. Since the machine can move the tape, it has the ability to read in information that it has written. Once it starts a computation, the machine continues to operate until it arrives at the halting state, which stops the machine's execution. If the halting state is never entered, then the machine will execute forever. The processor in such machines always executes some specific algorithm which operates the tape head and "informs" the machine when it has arrived in its halting state [BOAS90, SIPSE97]. The universal Turing machine (UTM) is a machine that simulates another Turing machine provided as input along with other symbols on its "tape" [SIPSE97].

There are many variations of the standard, single-tape Turing machine [BOAS90]. All of these variations differ from the single-tape Turing machine through time and space efficiency tradeoffs. However, since these machines are all machines that can be simulated by the UTM, they are therefore subject to the limitations of the UTM. Formal definitions of the universal Turing machine and various implementations can be found in [BOAS90, SIPSE97].

2.2.2. Limitations of the Turing Machine Model

Computer scientists, mathematicians, and philosophers have known that there exist problems whose answers may never be computed by a Turing machine. If a Turing machine is used to try to solve such problems, then it may never arrive at the halting state; —i.e., it may continue to run forever. These problems are characterized as *undecidable* [SIPSE97]. An example of this type of problem is,

> Given a computer program and precise specification of what that program is supposed to do (e.g., sort a list of numbers). You need to verify that the program performs as specified (i.e., that it is correct).
>
> — [SIPSE97]

Thus, if one is trying to prove that a given computer program halts, that too is also undecidable. This indicates that there are limits to what a Turing machine can compute. Since all of today's digital computers, outside of some research areas, are based on the Turing machine model, they too are so limited [COPEL99].

It remains an open problem as to whether there exists a computing device that can compute the answer to more functions than those of today's computers, which are based on the Turing machine model. Alan Turing first thought of such devices, known today as *hypermachines* or *oracles*, calling it an "O-machine" [COPEL99]. Furthermore, Kurt Gödel acknowledged to the American Mathematical Society that,

> "... on the basis of what has been proved so far, it remains possible that there may exist (and even be empirically discoverable) a theorem-proving machine which in fact *is* equivalent to mathematical intuition, but cannot be *proved* to be so, nor even be proved to yield only *correct* theorems of finitary number theory."
>
> — Kurt Gödel as quoted in [CASTI96]

This implies that there may exist machines that can solve more problems than today's digital computers. It has been noted that some researchers believe that the human mind is one such type of hypermachine [COPEL99].

2.3. Computing Architecture

Every computing device has an underlying model, i.e., *architecture*, from which it has been constructed. Today's digital computers actually employ a variety of architectures that are useful in computing including pipeline, processor arrays, cached systems, shared and disjoint memory, etc. Conceptually, these and many other architectures can be categorized by their behaviour at run-time (e.g., sequential, parallel, and distributed), temporally (e.g., synchronous, asynchronous, timeout policy), communicability (e.g., message serialization and

transport protocols, event notification models), device organization (e.g., queuing specifications, quality-of-service, knowledge representation), and device functionality (e.g., efficiency, correctness, and availability of algorithms). These categories are neither exhaustive nor mutually exclusive, rather they characterize different views of computing systems based on some selection criterion.

An implementation of a given computing architecture need not be a real, physical computing device (i.e., computing hardware), —it can be an abstract, logical one (i.e., a virtual machine). The difference between computing hardware and a virtual machine is that the former computes according to the laws of physics inherent with its design, whereas the latter is provided as encoded input to another machine that implements the architecture of another machine. In many cases, what is "real" *versus* "abstract" depends on the definition of what "physical" and "logical" mean for the user of such devices. For human beings, "physical" corresponds to objects that are understood in conjunction with at least one of the senses: hearing, seeing, taste, touch, and smell; whereas, "logical" corresponds to objects that can only be understood through cognition alone (i.e., without the physical senses). In practice, a *computer* refers to a physical machine (i.e., hardware) and a *virtual machine* refers to logical machine simulators (i.e., computer software). For clarity, the term *computing system* will be used to refer to any model or implementation that defines or performs computations (e.g., machine, human being).

2.3.1. Sequential, Parallel, & Distributed Computing

Today, there are three fundamental classes of computing architectures: sequential, parallel, and distributed. A sequential computing device is one which has a single processor capable of working on a single problem at a given time instant. If restricted to Turing machines, it is

synonymous with *Von Neumann* architecture found in today's "conventional" digital computers [BUYYA99b]. In contrast, a parallel computing device is one which can have more than one processor working together on one or more problems at a given time. Parallel machines therefore, have the potential to compute an answer faster by dividing up the problems-at-hand across two or more processors. Such systems are synonymous with *concurrent computer*, *multiprocessor system*, and the *multicomputer* [BUYYA99b]. Somewhat "in-between" sequential and parallel computing devices are distributed computing devices. Distributed computing devices are composed of smaller, usually independent, heterogeneous computing devices connected together by some type of communication network. Distributed computing allows for both sequential and parallel computing to occur within its network [BUYYA99b]. Technically, parallel computing devices are distributed computing devices with the distinction being that they are hard-wired, homogeneous processors within close proximity, i.e., less than 50 cm, to one another. A distributed device's processors are considered to be separated by larger distances, are entirely separate machines, cannot be assumed to be homogeneous, and are easily added or removed from the communication network. In any event, when sequential and/or parallel computing are used in a distributed computing network as a single computing system, that computing system's architecture is called *cluster computing* [BAKER99a].

Some might wonder why one should even bother researching parallel or distributed computing. The primary selection criterion necessary for computing depends on the *time* it takes to obtain an answer for a problem. If that answer arrives in a time interval considered acceptable for a sequential machine, then there is no need to utilize distributed or parallel resources. For many applications a sequential machine is sufficient and the most cost-effective

solution. A key reason that research into parallel and distributed computing is essential is because there exist problems considered intractable *without* the use of massively parallel/distributed computing devices. These problems have been called *Grand Challenge Applications* [ARGON99, BAKER99a, SILVA99, LI96] and involve massive amounts of computation over immense data sets. Another reason is the location of required resources, including time. If these resources can be found on a single machine, then sequential or parallel computing is usually sufficient, otherwise, a distributed solution will be necessary. This is an area where distributed computing stands out since it allows individuals that don't have such resources to efficiently utilize resources distributed throughout a network, such as the Internet. Since any distributed system involves, at some point, running sequential processes in parallel, research is primarily concerned with setting up an environment where that happens efficiently.

2.3.2. Flynn's Classification System

In an attempt to classify different sequential and parallel execution models, Michael Flynn proposed a classification scheme based on the number of instruction streams and the number of data streams that can be simultaneously processed [FLYNN96]. This resulted in the following architectural models:

SISD: Single Instruction, Single Data stream.
SIMD: Single Instruction, Multiple Data stream.
MISD: Multiple Instruction, Single Data stream.
MIMD: Multiple Instruction, Multiple Data stream.

Each data or instruction stream is independent of all other streams. Additionally, each instruction stream is a sequence of actions and each data stream is a sequence of objects.

2.3.2.1 The SISD Model

The SISD class of processors is the most common and has the least amount of parallelism; —it is a true sequential processor. Most of today's processors would be placed within the SISD model since they are based on Von Neumann's machine architecture [TANEN90]. Parallelism is often found in such processing systems, however, it is *internal* to the processor. With SISD computer systems, this parallelism is found at the hardware level as a distributed network of interconnected CPUs on the motherboard where techniques such as pipelining, instruction-level parallelism (ILP), and very long instruction word (VLIW) are typically employed [FLYNN96].

2.3.2.2 The SIMD Model

The SIMD class of processors includes array and vector processors and are ideal for efficiently processing data structures such as vectors and matrices. Array and vector processor architectures differ in their focus of parallelism. Array processors have multiple concurrently processing elements that operate on data elements. Meanwhile, vector processors have a single processor that operates on multiple data elements simultaneously. Due to their different designs, array processors obtain their efficiency by having large numbers of simple processing units, whereas vector processors depend on smaller data sets, pipelining, and high clock rates. Vector processors tend to have a higher latency period than array processors as the fetch of the data becomes one of the most important operations. Unfortunately, there aren't many SIMD architectural models due to their limited application base and market demands [FLYNN96]. Perhaps future dynamic topology construction techniques for hardware discovered through projects like the Oxygen project [AGARW99] will further the development and utility of the SIMD model.

2.3.2.3 The MISD Model

There has been little research done with MISD processing. As mentioned in [FLYNN96], abstractly the MISD model includes vector processors and the only machines implementing this model were perhaps those in the 1940s that used plug boards for programs.

2.3.2.4 The MIMD Model

MIMD processors are the most common form of parallel processors researched and developed thus far. Unlike the SISD and SIMD models, instruction-level parallelism is not hidden. Thus, the MIMD model permits the realization of the goals of multiprocessing and multiprogramming coding styles. In this model, it is possible to have different data storage models (i.e., policies) regarding which processors have access to specific data stores. Addressing these problems is important to maintaining data consistency and data cache coherency in computing machines. Today, data consistency is solved through a combination of hardware and software means; data cache coherency is almost always ensured through hardware means alone [FLYNN96].

2.4. Key Characteristics of Cluster Computing

There are six important characteristics that determine the utility of (distributed) computing systems: *resource sharing*, *openness*, *concurrency*, *scalability*, *fault tolerance*, and *transparency* [COULO95]. These are examined in the following subsections.

2.4.1. Resource Sharing

Anything that can be shared across computing systems amongst some set of users is a shareable resource. These resources include *time*, *space* (e.g., RAM, disk space, files), and *position/location* (of resources) (e.g., massively parallel systems, virtual private networks, the

Internet). There exist two models of resource management, the client-server model and the object-based model [COULO95]. The client-server model is the most common model in use today. It consists of a set of server processes that are each responsible for managing a set of resources and a set of client processes that request those resources. In this model all of the resources are held and solely managed by the server processes. The object-based model views every shareable resource as an object that is uniquely identified and can move anywhere within the network without changing its identification key.

2.4.2. Openness

The characteristic of *openness* refers to how a computing system can be extended. A computing system is said to be open when the *interfaces* of that system's machinery are published to allow the addition or transformation of its functionality by a third party without actually changing the systems construction. The more complete the published information is, the more open the system is considered to be. This characteristic is what enables the creation of interfaces that can permit heterogeneous hardware and software to interact in cluster computing [COULO95].

2.4.3. Concurrency

Concurrency is the appearance of several computing processes being run simultaneously [COULO95]. Hence, both sequential and parallel computing systems are capable of running concurrent processes. In a sequential computing system that is running N concurrent processes, concurrency is achieved by interleaving each process according to some time sharing policy. In a computing system with M processors ($M>1$), parallel processing allows for the possibility of reducing the total time to run those N concurrent processes by a factor of N/M. However, to achieve such time performance gains it is imperative to use algorithms

that efficiently synchronize accesses and updates to shared resources.

2.4.4. Scalability

Scalability refers to how well a computing system functions as the number of networked computing systems (i.e., *nodes* or *processors*) present as cluster size increases. Ideally the computing system and its computing methodology should not need to be explicitly changed when the scale of the system increases. The ability of a system to scale up to larger numbers of nodes is a measure of its utility for cluster computing. A system that does not scale up or down well will often require redesigning the entire computing system; —a task that is both costly and time consuming.

2.4.5. Fault Tolerance

Fault tolerance refers to a computing system's ability to detect and possibly recover from failures. A failure occurs when something happens that prevents a computation request from being correctly computed *as required* by its model. A failure can be manifested by errors in interpreting input or generating output, erroneous implementations (e.g., a software "bug"), or, by the lack of an established operating environment required for the computing system (e.g., water short-circuits a computer; a required network hub stops functioning). Unfortunately, not all faults are recoverable, however, as much as possible, it is desired that computing systems at least detect as many faults as possible.

2.4.6. Transparency

Transparency is the concealment from the user of separate real or logical components found in a computing system so as to allow the user to view the system as a whole, rather than a collection of component parts. There are eight types of transparency according to the ANSA

Reference Manual and the International Standards Organization's Reference Model for Open Distributed Processing (RM-ODP) (see [COULO95]): *access* (i.e., by providing a universal method to access any system component), *location* (i.e., by providing access to any system component without any knowledge of its location), *concurrency* (i.e., by providing concurrent access to shared information without interference from participating processes), *replication* (i.e., by providing multiple instances of the same data for performance and reliability gains without user knowledge of such), *failure* (i.e., by providing automatic fault recovery so that computations are completed), *migration* (i.e., by providing for the movement of components within a system without affecting the user or the system), *performance* (i.e., providing for the dynamic reconfiguration of the system as conditions vary without user interference), and *scaling* (i.e., providing for changes in scale of a system without further modification of its computing machinery) transparency. Specific circumstances dictate whether or not one desires a certain type of transparency.

2.5. Communication Models

2.5.1. Definitions of Data, Code, & Message

All computing systems may require input and may produce output. Anything provided as input to or as output from such systems is called *data*. Data are encodings of models that are assumed to be understood by either their senders or receivers or both. At times, it is necessary for the sender of some data to impart some "understanding" of a specific *model of computation* (i.e., procedure, function, operation, heuristic, etc.) to a computing system in order for that system to perform the desired task(s). When such models of computation are encoded, sent, and used by that computing system, they are termed *codes*. For example, a human programmer writes a program in a well-defined computer programming language

(e.g., C++) which is then fed to a compiler (i.e., a computing system) for processing. The compiler either produces machine-readable *code* output that can be *executed* by a computer or issues a report on errors in the original source code. Note that all implemented codes can be considered to be data, but, a datum can only be considered code if a computing system manipulates that datum as a model of computation. The latter distinction is important, e.g., while the computer programmer communicates C++ to the computer, it is not considered to be code by the computer until it is in machine-readable form and ready to be executed as a model of computation. A *message* is the data actually communicated from a user to a computing system and *vice versa* via their respective input and output interfaces. The term *message* differs from *data* in that the former implies communication between two logical or real entities, whereas the latter refer to the encoding itself.

2.5.2. Mobility of Data

Every computing system, whether abstract or physical, receives messages via their input interface and sends messages via their output interface. There are three different, commonly used ways of characterizing such communications: broadcast, point-to-point, and multicast (e.g., [BUYYA99b], [COULO95], [MPI95]). Further, such communications between components can be considered as (sub)networks in their own right.

2.5.2.1 Broadcast

A broadcast is a communication from an entity to an arbitrarily large set of entities. Broadcasts are one-way channels, always from a single source to many [BUYYA99b]. Examples of broadcasts include AM/FM radio broadcasts and writing to any of TCP/IP's broadcast addresses (e.g., 255.255.255.255).

2.5.2.2 Point-to-Point

Point-to-point communications occur between two entities and are implemented as two-way communications channels [BUYYA99b]. Examples include calling someone on a telephone and the use of Transmission Control Protocol (TCP) sockets (e.g., Telnet, FTP).

2.5.2.3 Multicast

A multicast is a communication to an entire group of entities. It differs from a broadcast in that the communications are not necessarily one-way and not all possible receiving entities may receive all communications packets sent (e.g., network congestion or failure, individuals leaving the multicast session, etc.) [BUYYA99b]. Internet action games, where people all over the Internet log onto a single server and play in the same virtual world, and teleconferencing can be considered multicast sessions.

2.5.3. Mobility of Code with Data

Since any useful computing system must interact with other systems, internally and/or externally, it is imperative that data are communicated using an understood language, or *protocol*, between the communicating systems. In distributed systems, the opportunity exists for the communication of code along with any required data, called *mobile code* [CARZA97]. By sending messages with code, the receiver can use such code to compute the answer to the problem on its own as it sees fit. When transmitting such messages, one imparts the knowledge resource (e.g., data) and/or the know-how (e.g., code) to the receiver of such. Research has been done by Carzaniga *et al* (see [CARZA97]) to categorize mobile code usage as paradigms to aid designing distributed applications.

2.5.3.1 Client-Server Paradigm

The client-server paradigm is one where a server offers a set of services to clients. The server has both the know-how and resources to satisfy client job requests. This paradigm is the most well-known and widely used of four (4) paradigms discussed by Carzaniga. Examples of this paradigm include the X Windows system and many of the standard Internet services such as the Domain Name Service (DNS) and File Transfer Protocol (FTP).

2.5.3.2 Remote Evaluation Paradigm

The remote evaluation paradigm involves the submission of a program or a set of executable instructions to another system with the appropriate resources to execute such. Consequently, there is a transfer of know-how from the submitter to the holder of the resources. In return the holder of the resources executes that know-how, whose output is often received by the submitter. From the submitter's perspective, the holder of the resources receives its know-how and must consider any costs that may imply. Similarly, the receiver of that know-how should consider the costs of malicious or false information that compromises that site's security or integrity.

2.5.3.3 Code on Demand Paradigm

The code on demand paradigm is the reverse of the remote evaluation paradigm. Instead of a user giving some know-how to another machine to perform a particular task, the user has the resources and simply asks the other machine to send the know-how for a particular need. Java applets on web pages are excellent examples of this paradigm. In asking for a particular web page, the user receives some information in return, which could include a Java applet that performs a specific task on the user's machine. Similar security and integrity concerns to those in the remote evaluation paradigm arise from the transfer of know-how in this

paradigm.

2.5.3.4 Mobile Agent Paradigm

The mobile agent paradigm is distinguished by an entity, called an *agent*, having the know-how to perform an operation but does not have the resources to complete that operation. Thus, the agent finds another entity that has such resources and *moves* itself completely to that entity's location so that it can complete its operation by using the other entity's resources. Such agents, often called *intelligent agents* (see [HAROL96]), have yet to be fully researched and utilized because of security and portability concerns. Such concerns can now be addressed by writing application software for the Java platform which has several classes designed for enforcing one's security policies when running third-party Java software (e.g., see [OAKS98]).

2.5.4. Topological Issues Concerning Computing Systems

Generally speaking, all computer programs are finite state automatons, which can be represented as graphs which are usually directed and acyclic [AHO86]. Each computer program consists of a set of states and a set of transitions from and to those states. Additionally, computer networks are also graphs consisting of a set of machines interconnected *via* various media (e.g., wires, optical fibre). Whereas computer networks are concerned with data communication, computer programs are concerned with computation. The resulting graphs in each case have topological structures that must be addressed. In order for any two computers within a network to communicate, an addressing infrastructure and communication protocol must be in place so that packets are reliably transmitted from one machine to the other. Similarly, in order to transit from one program state to another, a simulation machine must exist or be created to interpret a sequence of instructions to perform such. To provide various guarantees of performance and reliability, or, to facilitate protocol,

kernel, and machine architecture design, researchers often create systems that place constraints on the types of program or network graphs that are allowed to deployed on a computer or network. For example, nearly *all* of compiler theory (e.g., [AHO86] and [FIELD88]) is devoted to such topological concerns. A key problem in both computer programs and networks is the handling and assignment of addresses. While addresses are a necessity for identification and communication purposes, it is important that they are unique and are not too tightly coupled to any underlying software or data. These constraints have given rise to many different types of network structures, computer systems, and environments with varying degrees of flexibility, fault tolerance, and performance guarantees. It is not possible to review all of these systems, although some have been introduced under different guises above (e.g., broadcast, point-to-point, and multicast communication). Instead, the more recent research area of metacomputing systems (see [BAKER99b]) will be examined because it hybridizes the above concerns in an effort to create a more ideal, dynamic, and collaborative computing environment. Metacomputing systems research requires all of the knowledge of previous network and program topology *with* a lot of new ideas to solve many problems efficiently.

2.5.4.1 Metacomputing

The term *metacomputing* is thought to have originated with the CASA project in 1989 and was promoted by Larry Smarr, the NCSA Director, thereafter [BAKER99b]. A metacomputing system is, loosely speaking, a parallel computer where the nodes are dynamic as computational participants[1]. For example, if these nodes were people, a metacomputing

---

[1]The reader should note that this thesis presents a partial kernel for metacomputing purposes.

system is a ***network of networks*** of people (e.g., governments, corporations, societies, communities, families) that participate together in various ways to perform various tasks (e.g., govern, sell products, raise a family). Of course, over time individual persons, corporations, societies, governments, etc. join and leave networked individuals performing various tasks. Obviously, each individual performs their work concurrently relative to other individuals. Thus, these individuals can be viewed as a massively parallel "computing device". Although a precise definition of metacomputing has not yet been determined, metacomputers are analogous.

Metacomputing is essential to the long-term needs of humans. In a practical way, human computational needs are infinite, but our resources are finite. These resources include money, energy, and time. Hence, metacomputing aims to utilize as many free resources that are available at a given time by large sets of networked computers and it aims to do this efficiently with guarantees of cost and time. Ideally such resource utilization is seamless, fault tolerant, and supports all types of collaboration amongst its users. Practically, researchers have a long way to go before such lofty goals are met.

Metacomputing is currently represented by virtual computer architectures. Research focuses not on individual components of such systems, but rather on how such components work together as a resource unit. As noted by Baker and Fox in [BAKER99b], a metacomputer consists of the following four components: (i) processors and memory, (ii) networks and communications software, (iii) a virtual environment, and (iv) the ability to access data remotely. Further, they state that an implemented metacomputer,

> does not interfere with the existing site administration or autonomy; does not compromise existing security of users or remote sites; does not need to replace

> existing operating systems, network protocols, or services; allows remote sites to join and leave the environment whenever they choose; does not mandate the programming paradigms, languages, tools, or libraries that a user wants; provides a reliable and fault tolerance infrastructure with no single point of failure; provides support for heterogeneous components; uses standards, and existing technologies, and is able to interact with legacy applications; [and] provides appropriate synchronization and component program linkage.
>
> —[BAKER99b]

An ideal metacomputer implemented today would consist of middleware that works with existing systems in both traditional and non-traditional ways[2]. Baker and Fox include sufficiently general principles that metacomputing encompasses all knowledge and research in the computing sciences. While today's systems fall short of such an achievement, there are several metacomputing projects being undertaken. These include the Globus Project [GLOBU99] and Legion [LEGIO99].

2.5.4.2 Globus

The Globus Project [GLOBU99] involves the Information Sciences Institute of the University of Southern California (http://www.isi.edu/), Mathematics and Computer Science division of the Argonne National Laboratory (http://www.anl.gov/), and the Aerospace Corporation (http://www.aero.org/). The aim of Globus is to enable the construction and utilization of *computational grids*. A computational grid is a type of metacomputer that provides reliable access to high-end computational resources without regard to the geographical distribution of those resources.

The Globus Project consists of a kernel called the Globus Metacomputing Toolkit (GMT). The GMT may be viewed as a middleware network-enabled operating system as it is

---

[2]It may be that after many useful metacomputing systems have been researched and tested they will form a new basis for all operating systems.

responsible for things such as resource allocation, process management, communication services, authentication, remote data access, etc. Currently, Globus can interface with MPI (i.e., Message Passing Interface, [MPI95]), Java, C++, RPC, and Perl environments.

2.5.4.3 Legion

Legion [LEGIO99] began at the University of Virginia in late 1993 with its first public release at the Supercomputing '97 conference in San Jose, California. It is an object-oriented metasystem that strives to provide seamless interaction between geographically distributed computing resources, i.e., it is an attempt to provide to every user's workstation a single, reliable virtual machine. Legion is set up as an object-oriented system with both classes and metaclasses with the following attributes: (i) everything is an object, (ii) all classes manage their instances, (iii) users may define their own classes, (iv) with a core set object APIs to serve as its kernel. Legion emulates the PVM (Parallel Virtual Machine) and MPI APIs [BUYYA99a, LEGIO99]. It also interfaces with C, C++, and Fortran.

2.6. Temporal Models

It is essential at some point for computing systems to properly handle temporal concerns. Such is necessary to ensure job scheduling is performed properly, to observe job run-times and enforce policies, and, to provide applications with real and virtual synchronized clocks for a variety of applications including specific types of fault tolerance. There are two general categories of time classification: synchronous and asynchronous. These terms are correctly used when speaking of units of time on a specific clock, i.e., an event is said to be synchronous if it occurs at the same time as another event as observed on a common clock, whereas it is asynchronous when it doesn't occur at the same time [BUYYA99b]. These two terms are also used loosely with respect to computer software research and development and

refer to synchronous or asynchronous program event behaviours often implicitly. For example, the `synchronized` keyword in Java refers to the resulting code execution timing properties of the Java virtual machine's monitors [MEYER97]. Essentially, each `synchronized` object in Java serves as a simple virtual clock device.

Measured absolute and relative times are important to distributed and parallel computing because they are needed to establish reasonable concurrency control logic (i.e., job scheduling and control) within operating systems. Additionally, proper time measure is needed in many cases to establish algorithm correctness in distributed databases and to implement various types of fault tolerance and fault detection [COULO95].

2.6.1. Time Sensitivity

Callison [CALLI95] has proposed a *time-sensitive object* (TSO) model as an alternative to traditional (real-time) models based on constrained periodic and sporadic processes. The TSO model tolerates some types of system and timing errors, allows for a reduction in application complexity, and a possible increase in concurrency throughput. Traditional computing systems usually use periodic fixed time intervals to schedule process execution. This may lead to a variety of problems including odd timing interactions such as a low-priority process obtaining a time slice just when a higher priority process, blocked because of the lower priority process, requires access to a data stream.

Fundamental to the TSO model is that all data has a time period for which it is valid and outside that period, it is no longer valid. At one extreme are *single-interval transient* objects. These objects are created, live for a short period of time, and then are destroyed without their values changing. At the other extreme are those objects that are *immutable* and live as long

as the system which includes them. In general, most objects fall between these two extremes. For these objects, an allowance is made in the model to recognize the time interval that their values are valid. This interval is called the *validity interval.* The validity interval itself changes over time giving rise to a *current validity interval* and those objects that may store validity interval histories, called *multi-interval* objects. Multi-interval objects are either valid for a fixed time period after which they *will* change or they are expected to describe the latest time when some change of values *may* occur. The former is called a *periodic* time object and the latter is called *sporadic.* In the sporadic case, it is possible to have *error-producing*, *inferencing*, or *unbounded* conditions. An error-producing condition is caused by a failure of information to arrive by a specified time. One may infer correct timing behaviour based on the sequence of a set of events on which is based an *inferencing* condition. Finally, there may exist situations where no known time limit can be determined for an object (e.g., the time required for a networked computer system to come back up after crashing). This is the *unbounded* condition. These conditions allow the detection of errors and/or specific program states to which a system can be programmed to react. In general, time-sensitivity is important to metacomputing systems because they allow the possibly asynchronous events of joining and leaving of parallel computing sessions. The efficiency of such parallel computations would be highly dependent on attributes specific to real-time behaviours and is subject to models such as the TSO model detailed here. Thus, such models would be expected to be part of the design of a metacomputing system's kernel and architecture.

2.7. Some Cluster Computing Implementations

There are increasing numbers of cluster computing solutions available today. Most of these solutions are part of ongoing research projects; most are available free of charge and can be

put to immediate practical use. Three implementations will be briefly reviewed.

2.7.1. Message Passing Interface (MPI)

The Message Passing Interface (MPI) [MPI95, MPI97] is a middleware standard for message passing (e.g., function calls implemented *via* point-to-point or multicast communication, Linda [BENAR90]) whose main objectives are portability of developed code and high performance computing. The MPI standard is specified by a consortium of corporations, universities, and governmental organizations. The MPI standard contains hundreds of kernel API functions which are responsible for various types of communication (e.g., broadcast, point-to-point) and the administration of groups of such communications. Additionally, the MPI kernel permits blocking and non-blocking for nearly every class of kernel function to allow for maximum efficiency of code. Unfortunately, the MPI standard does not specify how different MPI implementations may communicate with one another and it does not allow the allocation of tasks to specific CPUs. Despite this, however, several implementations of MPI are available for supercomputing hardware as well as Unix servers and workstations. The MPI standard allows the programmer to develop with Fortran, C, and C++, which makes it popular with many researchers who use Fortran and may need to interface with legacy code.

2.7.2. Parallel Virtual Machine (PVM)

Parallel Virtual Machine [GEIST94], or PVM, from the Oak Ridge National Laboratory, permits a network of heterogeneous computers to be used as a single "virtual" computer, hence its name. PVM accomplishes this feat by requiring the user to run a kernel daemon on all of the machines within a given PVM machine. Consequently, programmers can write code using the PVM API calls to that daemon, thereby enabling portability. The PVM API calls are concerned with various types of communication methods (e.g., broadcast, point-to-point),

task control and coordination, and controlling the state of the virtual machine itself. Unlike MPI, PVM dictates an inter-PVM daemon protocol which enables different vendor implementations of PVM to talk to one another. For its flexibility, however, PVM has paid a small price in speed when compared to MPI, but it is available on nearly all Unix platforms and under Windows NT.

2.7.3. Virtual Distributed Computing Environment (VDCE)

Unlike the previous two implementations, Virtual Distributed Computing Environment (VDCE) focuses on creating a metacomputing system from the ground-up deployed through the World Wide Web *via* the Java virtual machine [TOPCU98a]. VDCE is a research project at Syracuse University and is intended for use across high-speed ATM networks. VDCE consists of three components: (i) a graphical programming environment, called the Application Editor, (ii) an Application Scheduler responsible for mapping resources to suit the requested task, and (iii) a run-time system which executes and manages all VDCE controlled resources. To facilitate deployment of the Application Editor to the user, VDCE takes full advantage of the web by authenticating users and subsequently deploying the Editor through that medium. The remainder of the system successfully deploys applications and returns results after computing across a set of nodes. When compared to the P4[3] message passing library, VDCE performed very well outperforming the P4 library in many instances [TOPCU98b]. Additionally the Application Editor, a GUI software construction tool, enables reduced program design and development times as well.

---

[3] P4 is a library for writing distributed shared-memory and message passing programs in C and Fortran [ARGON99]. It is a precursor system and served as a partial basis to early versions of MPICH [GROPP98].

CHAPTER 3.
The Modelling Cycle and Job Metamodels

All too often, the author has seen people discuss modelling as if it were merely constructing machine implementations. It is not simply just that. The most generic form of modelling concerns the modelling of modelling itself. Consequently, modelling is a potentially transcendental exercise that ideally can permit the inclusion of just about any meaningful discussion topic within its bounds. In science, one attempts to discover and validate truths from carefully controlled and reviewed observations. For scientists these truths are embodied as models (e.g., *via* mathematical argument, experimental observations, etc.) of the subjects of study. Within the scope of creating and implementing a modelling system, it is important to discuss aspects of metamodelling and the ability of machine models to process jobs. Such is discussed in this chapter, before presenting a partial solution towards that end in the following chapter.

3.1. The Modelling Cycle Metamodel

When one aims to build a kernel-architecture suitable for modelling systems, it is imperative to create and present a general model of modelling itself. Generally, the term *metamodelling* refers to a model of modelling, and similarly, the term *metamodel* refers to a model of a model[4]. All models of modelling may have common characteristics suitable for the creation of a universal metamodel of (meta)modelling. Just as the end product of modelling is the creation of application specific machine models and implementations, the end product of

---

[4] 'Model' and 'metamodel' are analogous to 'class' and 'metaclass' respectively (for the latter see [FORMA99]), however, a 'model' defines a machine, whereas a 'class' defines and/or implements an interface. Note that 'class' and 'metaclass' refer to specific concepts in the object-oriented paradigm; —'model', 'metamodel', as well as '(meta)modelling' are more general concepts.

metamodelling is the creation of modelling-specific machine metamodels and their implementations (i.e., models). This arises from the fact that all models are abstract definitions which define machines, thus all metamodels are therefore concerned with the abstract design of definitions of those machines. Of course, a metamodel itself also defines a machine as all metamodels are indeed models.

In his pursuit to discover a universal metamodel for any model, the author will propose below an initial set of characteristics for such a metamodel. All models of practical utility are initially conceived of (i.e., created) and likely are eventually destroyed. Hence, all models have a lifetime, from their individual moments of conception until their individual moments of death. It is useful to give an analogy here to biological organisms for they too have lifetimes. In biology, when one speaks of aspects of an organism's lifetime, one speaks of that organism's *life history* which includes things such as *survivorship* (i.e., the ability to survive over time) and *evolutionary* properties (i.e., the ability to adapt to new or changing environments; "survival of the fittest"; "evolution"). It is useful to consider a model's life history to enable research on how models should be designed in order to increase their survivorship and to account for their eventual deaths. To do so would hopefully allow researchers to minimize the cost in designing, implementing, and maintaining models. New and better models are discovered everyday, older models *evolve* to accommodate changes in their environment and the remaining unsuitable models perish in an environment in which they are of no utility. For example, consider a computer science research project where model design and implementation takes place amongst a group of researchers. Ideas and models will be conceived and will die throughout the time period of the project. The demands of the research project's goals, and that of its researchers, ***naturally select*** for those model designs

and implementations that are best able to satisfy such demands *within* the research environment. As a research project's environment is one of creating designs and implementations of models (i.e., metamodelling), it is imperative to recognize aspects of a model's life history. Such is outlined in Figure 1 below as a *life history cycle*, called the Modelling Cycle Metamodel.

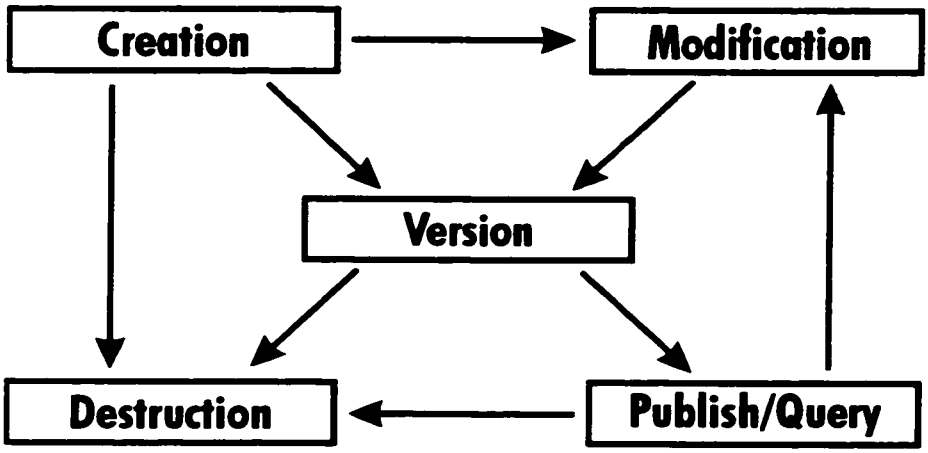

Figure 1. The Modelling Cycle Metamodel.

Figure 1 was created and defined by the author to better understand and communicate what happens to models *and* their implementations throughout their lifetimes. Doing so, aided the design of a suitable (partial) kernel-architecture in this thesis. The latter is important for if the kernel does not provide support for the above metamodel, then a key goal of this thesis (i.e., to address maintenance issues) has not been achieved. Figure 1 was conceived through the author's own modelling initiatives (e.g., object-oriented programming, analysis, and design patterns), general observations of how people use information (e.g., computer files, paper documents, etc.), and through his own knowledge and observations about research, cognitive, and modelling natural processes (e.g., experimental design, evolution of ideas, scientific method, and biological evolution). Additionally, Hammer *et al*, in a preliminary paper (see [HAMME96]), also address "lifecycle" phases as an important issue that must be addressed

in distributed object-oriented real-time systems.

A model's life cycle (see Figure 1) is composed of five phases: the creation (i.e., model conception), modification, version, publication/interrogative (i.e., **Publish/Query** in Figure 1), and destruction (i.e., model death) phases. The starting state of any model is the creation phase. For a model, creation is the act of defining its logical machine definition, whereas, for implementations, it is the act of defining an encoding of some model. After its creation, a model or implementation may be modified. If such happens, then its corresponding machine or encoding definition has somehow been altered. In any event, at some point a model or implementation should be in a state to be recognized by other models or implementations (i.e., the **Version** phase in Figure 1), unless it perishes first (i.e., the **Destruction** in Figure 1). Having the **Version** state in the Modelling Cycle will allow a formal definition of model evolution to be defined in the future in terms of version identifiers that will be associated with models. Further, such identifiers would serve to identify individual models within a population of models (i.e., a modelling system). Additionally, by passing through the **Version** phase, models and implementations are allowed to be published and interrogated after which it is accessible by an appropriate group of users as dictated by some usage metamodel that is left unspecified here. Once published, a model or implementation may be modified to give rise to new versions of themselves. Sooner or later, however, it is possible for any model to be destroyed after which it no longer exists (i.e., the **Destruction** phase in Figure 1). The reader should note that such destructions may not only arise from within the modelling system, but, may also be manifested by violations of a system's axioms (e.g., the loss of power to a digital

computer housing such a system). As such, it is possible for all models to die[5]. Consequently, no model can wholly control all aspects of its life cycle.

Although somewhat off-topic, it is worthwhile to engage in some discussion of some intriguing biological questions and concepts applied to the domain metamodelling. It may surprise the reader that biological issues related to taxonomy[6] (e.g., "What criteria constitutes species differentiation?"), sociology[7] (e.g., "What set of behaviours define a community?", "How do communities interact?"), and phenotype (e.g., mimicry of physical appearance) are all relevant in the domain of metamodelling. For example, the above can be applied to metamodelling by noting the following: a biological *species* is the same as *(model) type*[8], *taxonomy* is a hierarchical extension of a *formal type and naming schema of models*, *sociology* is a study of *dynamic or run-time behaviours of models*, and *phenotype* is equivalent to a model's *interface* (e.g., an abstract class in object-oriented programming). Further, it is interesting to note that biological evolution *directly* selects for organisms of a specific

---

[5]Except God! Atheists and agnostics may prefer substituting the notion of God with the notion of a metamodel that defines and, at least partially, implements all that has existed, exists, and will exist, or potentially so, should such a metamodel exist, as one metamodel, or as several cooperative metamodels.

[6]i.e., the hierarchical classification of organisms based on a specific set of criteria.

[7]i.e., the communications structures between interacting organisms.

[8]Here, *type* is used in the computer science sense. Specifically, *type* refers to a fundamental classification of what something *is*. It is interesting to note that there is no universally applicable definition (i.e., model) in biology of what a species *is*. *Type* has similar problems in metamodelling systems (e.g., Paul's *list* type is a "doubly-linked list" whereas Susan's *list* type is a "hash table"). Due to timing considerations, this issue has been left out of this thesis.

phenotype[9] [RAVEN89], —only *indirectly* selecting for that organism's expressed genes. The author hypothesizes that the same will be found true when observing evolution in metamodelling systems (i.e., where an *organism* is a model or implementation, its *genes* are its definition, and its *phenotype* is expressed as a **Broker** presented in the next section). An example towards supporting this hypothesis is the existence of mimicry of object-oriented interfaces found in the *Adapter* and *Proxy* design patterns in [ALPER98] and [GAMMA95]. Such mimicry is argued here as an evolutionary response to the selection pressures of class interface redesign. The *Adapter* and *Proxy* patterns allow existing models and implementations (i.e., object-oriented classes and their instances) to survive longer (i.e., increase survivorship) in new environments. With such interesting questions and examples, it is the firm belief of the author that it is absolutely essential for long-term metamodelling research to fully examine metamodelling relevant structures such as these. These structures should be compared to already well-known and well-studied ones found in biology, chemistry, physics, economics, and many other fields of research and practical study.

Thus, the Modelling Cycle Metamodel is proposed to describe the transition of models from their respective births until their respective deaths. Implementing such a metamodel within a real modelling system, however, will have to wait until the author creates a sufficient metamodelling system wherein such can be defined and experimented with. Until then, the author can only engage in abstract debates and dissertations of the cycle. Nevertheless, it has been presented herein to address the long-term ideal goals of this thesis to aid the development of metamodelling as a formal research field.

---

[9]An organism's phenotype is its physical appearance as seen by another individual or selection pressure. *Note:* it is an *indirect*, outward manifestation of an organism's genes *within* an environment. Examples include colour blindness, mid-digital hair, size, etc.

3.2. The Job Metamodel

The practical utility of any modelling system, as far as this thesis is concerned, is its ability to compute. That is, a model defines a machine that computes something given an appropriate message stimulus. The coupling of a model with a specific message stimulus is a job request that performs a specific computation over that datum (i.e., the message) by some encoding (i.e., implementation) of that model. Ideally, the user that made the job request obtains a job result within some set of job requirements, although this need not be the case. If the former happens, the job is considered successful, otherwise the job is considered unsuccessful. If one is to construct, understand, and research a metamodelling system that can potentially run any appropriate job applied to it, it is essential to view how a user interacts with any abstract or real machine. This section provides an overview of such.

3.2.1. Jobs and Their Users

For a model and its implementations to be of practical utility, there has to be some set of external users of such. From the user's perspective, a model or implementation appears to be a machine that may compute (e.g., the **Computing System** in Figure 2 below) something (i.e., a job result) given some initial stimulus (i.e., a job request). To communicate the job request to the machine and to then comprehend the job result returned, the user will interact with a set of devices on/in the machine that *brokers* each of these communications. How the machine actually performs its computation is not irrelevant to its user(s)[10]. An illustration of a user interacting with a computing system is shown in Figure 2 (see page 38).

---

[10]What *is* relevant to the user is the selection of an appropriate machine to perform the desired computation. Further detail of any selection methodologies and pressures are beyond the scope of this thesis.

Figure 2 shows a set of users interacting with a set of devices, collectively labelled as the **Broker**. All that the user ever "sees" of the **Computing System** is the **Broker** component, the job requests the user(s) send (i.e., the arrow from **User(s)** to **Broker** in Figure 2), and the job results the user(s) receive (i.e., the arrow from **Broker** to **User(s)** in Figure 2). The remainder of the **Computing System** is a black box of an unknown number of processing units that perform the task individual users *think* it is performing (i.e., the dashed arrow between **Broker** and the numbered **CPU** units within the machine in Figure 2). In general, a computing system need not return any job results or even do any processing whatsoever. This too can be considered a computation, —such as the relatively common computer NOP[11] instruction. As the definition of a user, model, and implementation define machines and their operations have been discussed earlier in this thesis, the only remaining component of the Job Metamodel that requires further discussion is the **Broker**.

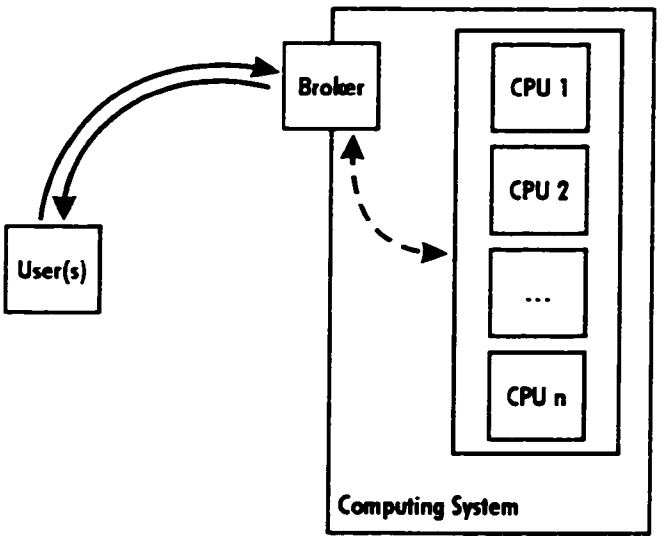

Figure 2. Illustration of a user interacting with a computing system.

[11] NOP is a common Assembly language instruction corresponding to that machine's "No OPeration" machine instruction. The term is used loosely here.

The **Broker** provides all of the mappings between a computing system's external input and output *interfaces* and their corresponding internal machinery. All machines have internal components designed to accomplish a set of tasks and some additional components designed to allow external users to effect specific computations on such machines. As mentioned earlier, the internal components of a computing system are irrelevant to the user, however, the components that represent the **Broker** are relevant. This is because it is through the **Broker** that the user interacts with using some set of conventions (i.e., protocols) in order to communicate with the computing system machinery. Such user-computing system interaction is called *brokering*, hence the term **Broker** used here. For example, if computing system is defined as an electrical circuit composed of a light bulb and a single pole-single throw switch, it is the switch and the light filament that comprise the **Broker**[12]. The remaining components, the wiring, electrons, and other materials in such a system are all internal machine components. Therefore, as the **Broker** provides mappings between the external (i.e., user) and internal (i.e., computing system) environments, it must be considered to be a translator or compiler between the protocols used in the two environments. These protocols need not be the same (e.g., the physical motion of flicking a light switch is an external language which is mapped internally to either a flow or non-flow of electrons, *an* internal language). This ends the overview of the Job Metamodel. The task of further detailing the Job Metamodel is the primary focus of the remainder of this thesis document. It begins with an example illustration of a user using a computing system in the next section.

---

[12]It is assumed that the user of such a system is a human; that when the switch is in its closed state, the circuit is completed wherein electricity flows and causes the filament to emit photons; and, that when the switch is in its open state, the circuit is broken and the light filament doesn't emit photons .

3.2.2. An Example

It is important to illustrate how a user would use a system based on the Job Metamodel so the reader may better appreciate the (partial) kernel-architecture design proposed and presented later in this thesis. The example is illustrated in Figure 3 below. Figure 3 is identical to Figure 2 except that it now includes more detail to aid in understanding the example.

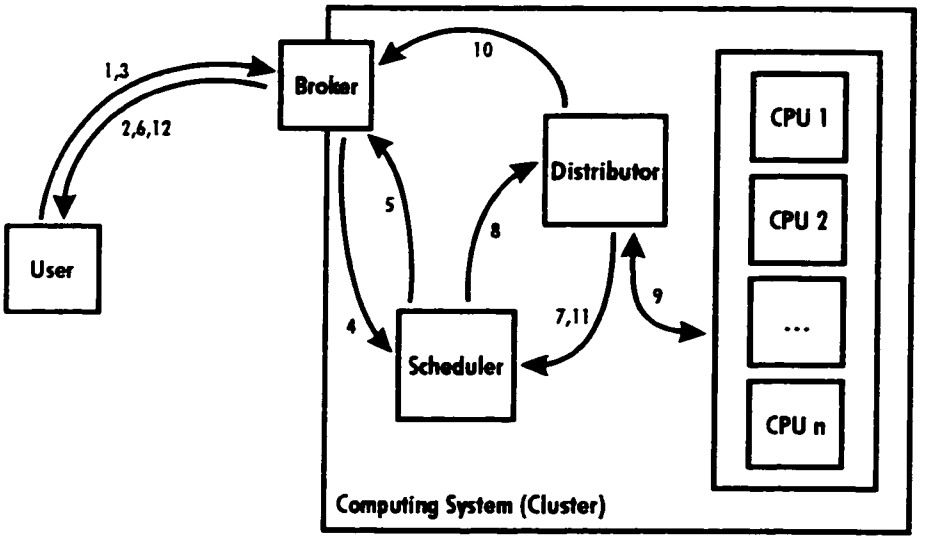

Figure 3. An example illustration of job dataflow through a computing system.

Consider the scenario where a single user wants the said computing system to perform a job. For simplicity of presentation, it is assumed that this job will run successfully. The above figure illustrates the message flow, represented by numbered arrows, from the user's initial job request until the user receives the job results. The relative temporal sequence of each message is noted by increasing numbers next to each arrow. In addition, the computing system is composed of a **Broker**, defined earlier, a **Scheduler**, that schedules machine processing jobs, a **Distributor**, that controls and manages job distribution, and a set of processing machines units, labelled as **CPU 1** through **CPU n**. The submitted job proceeds, therefore, with the following twelve (12) chronological steps below whose numbers correspond to the numbered

arrows in Figure 3:

1. The **User** provides a message as input to the **Computing System**'s **Broker**.
2. The **Broker** responds with an acknowledgement that it has accepted the message provided in the previous step. Such an acknowledgement may include returning a unique *job request identifier* to allow future messages to reference the submitted job.
3. The **User** asks the **Broker** to start the job within the next three (3) hours.
4. The **Broker** then forwards the request from the previous step to the **Scheduler**.
5. The **Scheduler** finds an available time slot within three hours to run the job so it tells the **Broker** that its request is satisfied.
6. The **Broker** then tells the user that the request, from step 3, is satisfied, as determined in step 5.
7. At some point in time, the **Distributor** would have or will notify the **Scheduler** that is has finished some set of jobs. This is not a prerequisite step for step 8 to occur.
8. When the time arrives to run any scheduled job (e.g., the one in step 5), the job is sent to the **Distributor**.
9. The **Distributor** performs whatever is necessary to execute the job across some set of machines (i.e., **CPU 1** through **CPU n**) as it sees fit or as specified within the actual job request. When complete the **Distributor** has the job results.
10. The **Distributor** then sends the **Broker** the job results.
11. The **Distributor** tells the **Scheduler** that the job has been completed.
12. The **Broker** communicates the job results to the **User**.

This example is not unlike a user going to a web site and requesting some data (i.e., step 1), to which the web site responds that the request has been approved (i.e., step 2), and then at some scheduled time, the web site computes and then sends back the requested data to the user (i.e., steps 4 and 7-12), say *via* email. The user is not aware of how many machines are actually used with such requests; —the user only interacts with the web site which serves as a **Broker** for such requests. **Brokers** also permit the implementation of security and external usage policies of a computing system and the ability to *encapsulate* or hide the internal components of a computing system. This serves to facilitate the computing system's utility as well as to protect it, just as encapsulation and strong type-checking does for object-oriented systems.

One should note that the twelve steps can be broken down into three phases, the job request phase (i.e., steps 1 and 3), the computational phase (i.e., steps 4-5 and 7-11), and the job result phase (i.e., steps 2, 6, and 12). One could view this example as a user's single job or as two jobs depending on whether or not steps 1 and 3 are considered as single or separate jobs. While such distinctions may seem arbitrary, they may be important to consider during the implementation of model machinery to establish user-required guarantees such as computational efficiency, security, etc. Further, the user need only know what a given implementation's underlying model is and whether or not it is correct and reliable. This is the reason that only job requests and job results involve interaction between the user and the computing system, —the job computation phase occurs entirely within the computing system itself, independent of any direct user interaction. If direct user interaction with anything except the Broker is permitted, then the user would need to fully understand the internal implementation of that computing system before interacting with it in a meaningful way. This

is not desirable as it creates more than one point of interaction with the model of interest complicating the definition of the use of any devices found within the **Broker**.

CHAPTER 4.
The Job Metamodel in Detail

At the heart of the Job Metamodel is the need for a user to interact *via* a broker with a computing system. Fundamentally, this requires a machine model that inherently represents the broker and its operations where job requests and results can be communicated while allowing job computations to be performed. Clearly, such a model must have both external and internal interfaces used by the user and the machine respectively. This chapter details precisely this, a partial modelling infrastructure that must be represented when constructing any metamodelling system. The infrastructure defined herein defines *abstractly* how computations are to be performed.

4.1. Virtual Machine Node Model

To sufficiently represent a computing system, one needs to couple an appropriate broker interface with a hidden method (i.e., a model or implementation) of computation. The broker interface is responsible for capturing job requests and releasing job results, whereas the model of computation attempts to use those job requests to compute some type of job result. Taken together, the broker and computing machinery comprise a general machine model called a *node* (i.e., a **Computing System** as presented in the previous chapter). If such a node is implemented abstractly (e.g., in software) then it is considered to be *virtual* otherwise it is considered to be *real*. It is possible to compose a series of nodes together to form machines by connecting the appropriate job result flows to the proper job request sinks. By composing and connecting machine nodes together, arbitrary larger machines are possible from a well designed set of primitive machines (i.e., nodes). Ideally, every node should be able to be represented by a common model so as to facilitate the design and implementation of nodes

independent of any context-sensitive brokering or computational attributes. The virtual machine node model presented in this section meets these ideal requirements. It is illustrated in Figure 4 below.

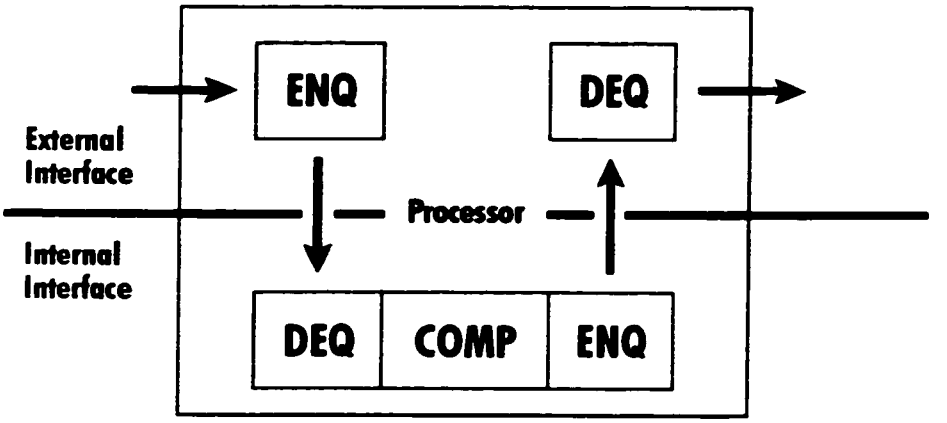

Figure 4. The Virtual Machine Node Model.

Each virtual machine node contains three types of internal components: (i) an interface to insert job requests into a queue, called the *enqueue* interface (i.e., **ENQ** in Figure 4), (ii) an interface to remove job results from a queue, called the *dequeue* interface (i.e., **DEQ** in Figure 4), and (iii) a processor where computations are defined to occur (i.e., **COMP** in Figure 4). The enqueue and dequeue interfaces together form a queue. The actual node queuing interface that the user sees is shown as the **External (Machine/Node) Queue** in Figure 5 below. As it completely found in the user-accessible, external interface of the machine node, this (abstract) queue is what has been referred to as the **Broker** in the previous chapter and, in this chapter, the *broker interface*. The **ENQ** and **DEQ** devices within this interface are, themselves, virtual machine nodes that *enqueue* job requests and *dequeue* job results respectively. This is highlighted clearly in Figure 5.

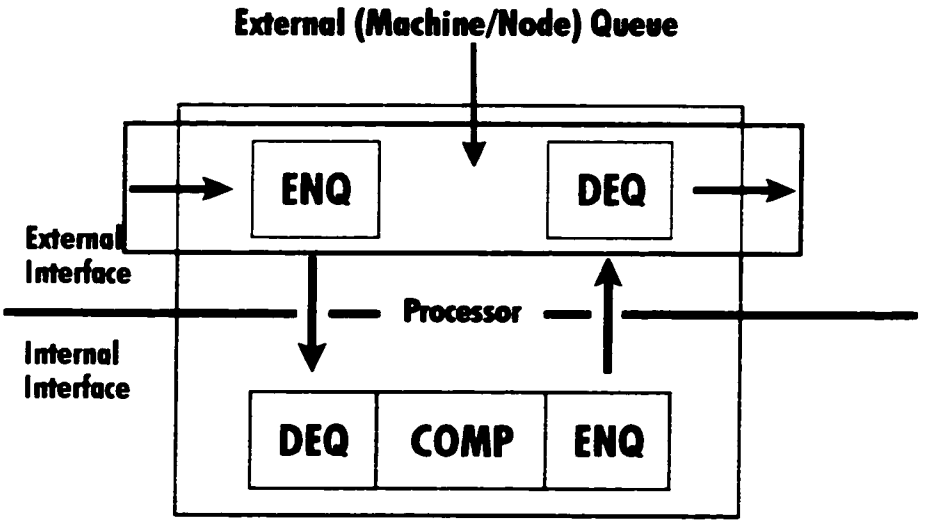

Figure 5. The Broker: The virtual machine node's external queue.

The broker interface should be viewed as a *transforming priority queue*, a prioritized sequence of incoming job requests that are possibly transformed into outgoing job results. The transformations employed on job requests are simply the computations that the machine performs on such. This type of *priority queue* is in contrast to traditional priority queues[13] that do not alter the state of any elements while they are in the queue, —only their temporal ordering of extraction from the queue may be altered as compared to their insertion. The reason the broker interface must be *transforming* is to accommodate the processing of job requests and the generation of job results. In general, a job result is *not* what was enqueued as a job request. Additionally, the queue must be somehow *prioritized* as it is possible that incoming job requests may not be immediately processed upon arrival, forcing such requests to be placed on hold or *effectively ignored*[14]. This is necessary since a machine cannot wholly control all of the aspects of its own existence (i.e., the Modelling Cycle Metamodel in the

---

[13]i.e., as known in the field of computer science.

[14]The term *effectively ignored* is used to cover any situation whereby a job request does not cause any corresponding computed job result to be output from that machine node; i.e., the machine node is permitted to fail, by accident or by design.

previous chapter); —a model can neither create itself nor completely *prevent* its destruction by a third party.

The queue that the user sees as the broker interface is not a single queue. The broker interface is really composed of two separate transforming priority queues: the incoming job queue in Figure 6,

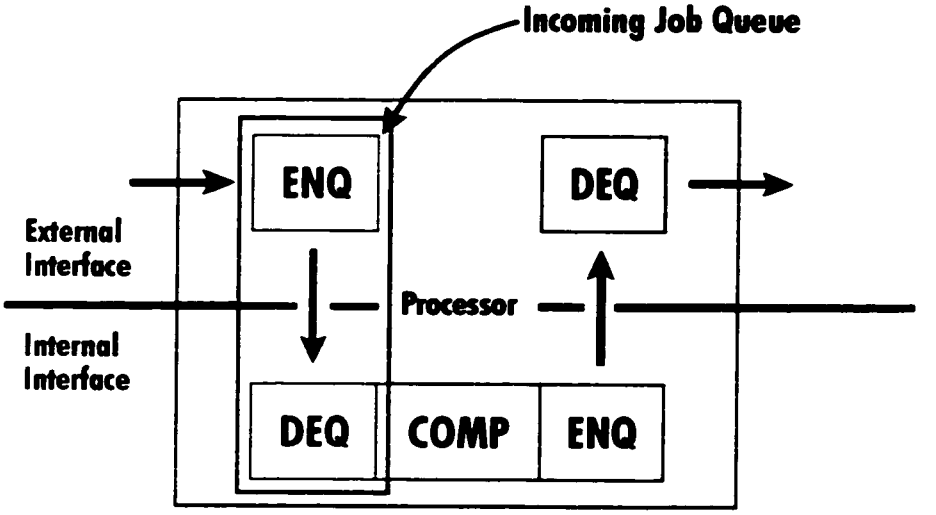

Figure 6. The virtual machine node's incoming job queue.

and the outgoing job queue in Figure 7,

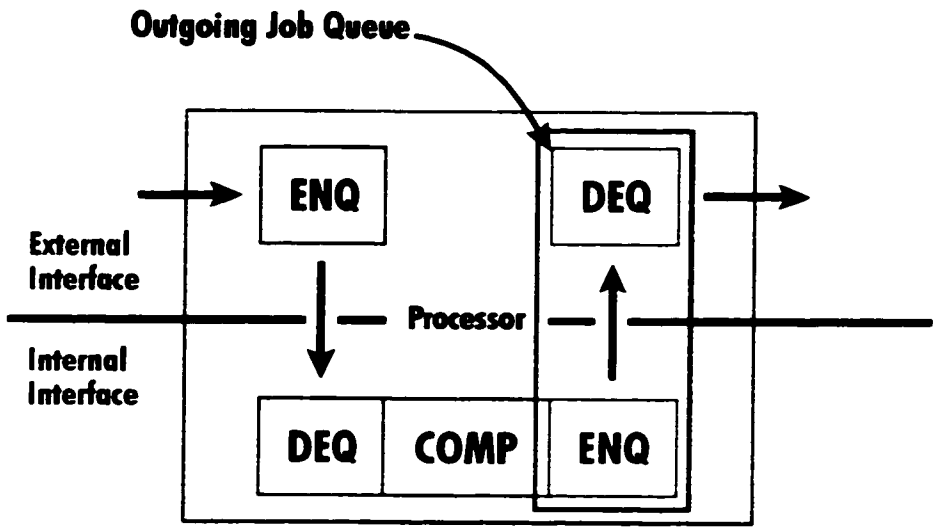

Figure 7. The virtual machine node's outgoing job queue.

Thus, the enqueuing portion of the incoming job queue and the dequeuing portion of the outgoing job queue represent the broker interface. These two queues are linked by the processor which handles the items dequeued from the incoming job queue and produces the items enqueued on the outgoing job queue. It should be noted that it is the processor that connects a dequeuing interface with an enqueuing one. The latter is significant as any node can be viewed as a processing unit. This implies that a computing system may have nodes that serve to connect together other nodes in order to establish broadcast, point-to-point, and multicast communications. Further, the incoming and outgoing queues are machines in their own right and each may be implemented by other virtual machine node instances. Such a nesting of nodes within a node is presented in Figure 8 (see page 49).

Since nothing has been said about how, when, or what type of computations are performed, the presented node model is sufficient to serve as a fundamental architectural element of a computing system. Further, the Virtual Machine Node Model is distinct from the universal Turing machine; —Turing defined his machine as capable of only computing any calculation that a mathematician could, as long as it was performed by a specific algorithm with limited

time and energy in an unintelligent, yet disciplined, manner. The Virtual Machine Node Model considers nothing other than the *broker interface* by which computations *may* be performed (i.e., job requests and results). Thus, it may be possible to create an implementation of the Virtual Machine Node Model that could perform computations in an intelligent, even undisciplined manner. This allows for the possibility for wider scope of modelling opportunities since the Virtual Machine Node Model, while distinct from the universal Turing machine, could easily accommodate Turing's model. Thus, the proposed Virtual Machine Node Model is conjectured to be entirely suitable for use as a fundamental architectural element of modelling systems.

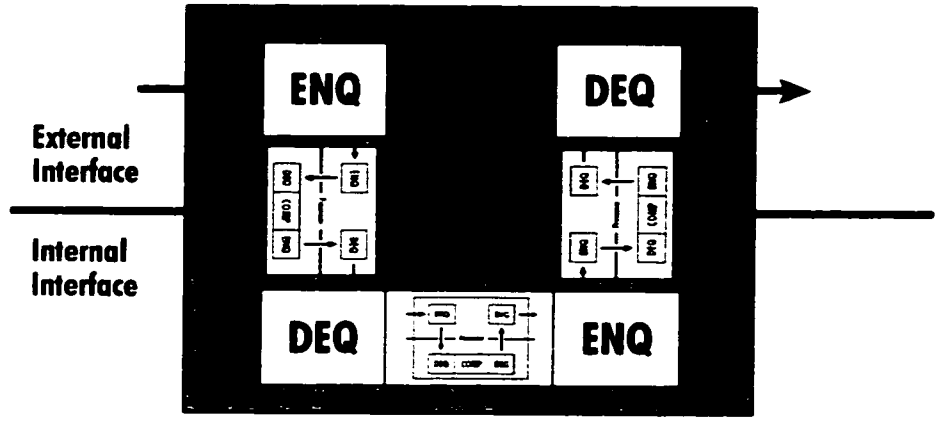

Figure 8. The nesting of virtual machine nodes within a node.

4.2. Aspects of the Job Metamodel's Design

There are many aspects of an ideal Job Metamodel not included in the presentation above. These include fault tolerance, transaction processing, policy enforcement (e.g., security, usage, authentication, and encryption), queue management strategies, machine and job administration, resource acquisition issues, concurrency issues (e.g., deadlock avoidance, job distribution, and job scheduling), topology and architectural issues (e.g., of machine construction, job deployment, processor type: SISD, MISD, SIMD, MIMD, etc.). Because of

their importance in computing, all of these topics are warranted in a complete dissertation of the Job Metamodel. However, here the Job Metamodel is incomplete as it does not define such. This is one reason why this thesis provides only a partial solution to the goal of establishing a ideal metamodelling system. The Job Metamodel presented here is not limited, —its design actually *allows for* the implementation of aspects of an ideal Job Metamodel.

4.2.1. Allowances for Other Job Metamodel Details

Just as it would be foolish to define a Job Metamodel that is not compatible with the Modelling Cycle Metamodel, it would be foolish to define a Virtual Machine Node Model (VMNM) that is known to not be compatible with a completed Job Metamodel. Each of the above aspects of a Job Metamodel can be implemented within the abstract design of the VMNM. As each component of the VMNM can be seen as real or abstract machines in their own right, it is possible then, with the proper models of fault tolerance, transaction processing, policy enforcement, etc. defined, to implement all of the aforesaid aspects as contained machines with a VMNM. This is especially true as all machines are have effectively three transforming priority queues and a processor. Thus, in a set of VMNM implementations, one can implement the aforesaid aspects within any set of transforming priority queues and/or set of processors as is proper. Some of these aspects will dictate specific interface requirements of job data, however, as such are either implementations or models themselves, they constitute abstract or real machine realizations that can be considered to be represented by the VMNM. Therefore, it is argued that the Virtual Machine Node Model can be used to satisfy the aforesaid aspects.

4.2.2. The Enqueue and Dequeue Interfaces

The enqueue and dequeue interfaces only have the ability to enqueue and dequeue job data

respectively. In this thesis, the author represented these interfaces similar to that of a traditional queue known to computer scientists (i.e., see `Enqueue`, `Dequeue`, and `Queue` on pages 79, 78, and 83 respectively). The traditional queue was chosen because it hides the queuing nature of the internal processor from the Virtual Machine Node Model's design; —i.e., it allowed all nodes to have the same broker interface. Arguments for a non-traditional queuing interface can be put forth arguing that there should be a "Parallel" Virtual Machine Node Model that permits enqueuing and dequeuing across a *set* of broker interface queues to directly support parallel data flows. The latter would add state information to each queue that determines correct queue utilization and it ties an *encoding* of a queue identification model (e.g., two's complement integers) to the Virtual Machine Node model. Since the addition of such information complicates and limits how the Virtual Machine Node Model could be implemented, the debated "Parallel" Virtual Machine Node Model was seen as inferior to the presented Virtual Machine Node Model.

The Virtual Machine Node Model may appear to some to support only a single data stream, however, it can actually support multiple data streams. This can be accomplished by specifying and using job data interfaces that support multiple data streams (e.g., an array of data streams). The enqueuing and dequeuing of such job data can be seen as a *parallel transmission* of a tuple as is often required by machines that support multiple data streams. Such nodes would be considered vector processors. One can also create a broadcast node that upon receiving a job request, distributes the received data to a set of nodes that compute in parallel. When complete those nodes can transmit their results back to be accumulated. These nodes can be nested, as a group, within a node to form an array processor. Thus, multiple data streams can be handled by the Virtual Machine Node Model.

Multiple instruction streams can also be handled. An instruction stream is simply a data stream containing instructions on how to perform a set of tasks. Multiple instruction streams can be achieved by transmitting code to a set of nodes that will execute the code they each received. Further, the single and multiple instruction and data stream models can be combined to produce systems corresponding to each of Flynn's four models. Thus, the presented Virtual Machine Node Model can be utilized to support all of the machine architectures addressed by Flynn.

4.3. A Java Implementation

In the end, it is important to demonstrate a model by implementing it. The Job Metamodel was implemented as a preliminary microkernel (simulation) in the Java virtual machine. The `public` and `protected` Java interfaces that implement the kernel-architecture design of the Job Metamodel are found in Appendix A (see page 70). While some classes are present to facilitate the use of the kernel within the Java runtime environment, all of the important models are defined as abstract classes/interfaces from which implementations are derived. Although the designs presented in Appendix A are specific to Java, the involved classes/interfaces can be easily ported to other languages and platforms. It is, perhaps, best to map all of the relevant elements from the Job Metamodel to their corresponding Java classes and interfaces defined in Appendix A. These mappings are defined thus:

Job Metamodel Term	Java Class/Interface Name	Page #
virtual machine node as a model	Queue	83
virtual machine node as implementations	VM, VMBus	97, 101
enqueue interface (i.e., **ENQ** in Figure 4)	Enqueue	79
dequeue interface (i.e., **DEQ** in Figure 4)	Dequeue	78
computing system processing unit (i.e., **COMP** in Figure 4) as a model	Processor	93

All of the remaining classes and interfaces in Appendix A are present to construct a meaningful implementation of the presented Job Metamodel's kernel-architecture. This implementation and its demonstration are detailed in the next chapter.

CHAPTER 5.
The Implementation & Its Results

In order to ensure that a model is of practical utility, it is necessary to construct an implementation of it designed to test various aspects of that model. To this end, an implementation of the Job Metamodel (i.e., a Job Model) presented in the last chapter was created and then implemented. Two example machines were also constructed to demonstrate usage of the Job Model's implementation. This chapter outlines various aspects of the Job Metamodel's partial implementation and gives a high-level overview of the demonstration programs created to show a physical realization of the metamodel.

5.1. The Selection of an Appropriate Implementation Environment

The selection of a suitable run-time environment and machine platform to implement the Job Metamodel was not an easy process. Since the digital computer using object-oriented technology is an ideal implementation platform to simulate the Job Metamodel, it was selected to do precisely that. This meant that the Job Metamodel implementation would be written as computer software.

When writing computer software, there is a question of how much does one base a product upon a third-party product's interface. Ideally, if one has the time, one would simply attempt to create a complete environment for metamodelling from first principles. Thus, the author had essentially two choices: to use an existing system as a basis or to write a virtual machine from the ground up. Either way, the implementation must allow for some representation of models and implementations that would allow their definition, creation, and inheritance of

attributes[15]. In the end, the Java language and platform was used to implement the concepts presented in this thesis.

Java, as a language and platform, has a number desirable features that would serve to facilitate an implementation of the Job Metamodel to prefer it over other general purpose programming languages such as Smalltalk, C, C++, etc. and their respective environments. Those features are: (i) its compiled form can be executed on virtually any existing computer, (ii) it is able to interoperate with web browsers within a secure, easy to download environment for the user, and (iii) it is object-oriented, supports concrete and abstract classes (i.e., models), multiple inheritance of interfaces, and class serialization (i.e., to easily transmit data between nodes when necessary). Other excellent, easily available, general-purpose computer languages and platforms (e.g., Smalltalk) could not satisfy such requirements without a significant investment in compilers for different platforms and writing code specific to each platform to perform tasks such as network socket I/O and concurrency (e.g., threads). Thus, Java was selected as the implementation language and platform.

5.2. The Kernel Implementation

The complete `public` and `protected` interfaces for all of the classes that serve as the partial kernel implementation of the Job Metamodel are listed in Appendix A (see page 70). The previous chapter merely pointed out which Java classes/interfaces correspond to specific Job Metamodel terms. However, to fully appreciate the implementation, some further discussion is required to explain how the kernel functions.

---

[15]For reasons why inheritance should be considered essential see [FORMA99].

5.2.1. VM and VMBus

There are two implementations of the Virtual Machine Node Model: `VM` and `VMBus` (see pages 97 and 101 respectively). Instances of both classes have installed processors (i.e., classes derived from `Processor` on page 93) and queue (i.e., classes derived from `Queue` on page 83) instances usually set via object construction. To facilitate machine development, testing, and deployment, each `VM` object defaults to using `ProducerConsumerMonitor` (see page 85) instances for its incoming and outgoing job queues and a `ProcessorForAlgorithm` (see page 95) object if an `Algorithm` (see page 87) instance is passed into its constructor. Meanwhile, each `VMBus` object is hard-wired to use `ProducerConsumerMonitor` instances for its queues, and, its processor type must be an instance of `ProcessorForDispatcher` (see page 96). Together, these two Virtual Machine nodes allow one to dynamically create and then simulate a machine.

Alone, a `VM` instance, as a consequence of its use of `ProcessorForAlgorithm`, will *automatically* dequeue and process information sequentially as job data is enqueued[16]. The type of processing it performs depends on which `Algorithm` was passed into the constructor of that `VM` object. While a `VM` is customizable by using custom `Processor` and `Algorithm` types, a `VMBus` node only accepts different types of `Dispatcher` objects. This is a consequence of `VMBus` using a `ProcessorForDispatcher` instance to allow one to string together `VM` nodes into a state machine. In general, each `VMBus` node is a machine that *automatically* forwards queued outgoing job results to other incoming job queues. By way of contrast, `VM` nodes do not have the ability to *automatically* forward requests from its incoming queue to other machine's incoming queues because `ProcessorForAlgorithm` doesn't provide such behaviour. If this machine were represented

---

[16]A different type of `Processor` could be used to provide different types of concurrent behaviour. With `ProcessorForAlgorithm`, the jobs are processed in a first-come, first-served (i.e., FIFO) fashion.

as computer hardware, each VM node would be a chip on a computer's motherboard, whereas, each VMBus node would be that computer's bus; —the wiring that connects together those computer chips. Just as the laws of physics governing current flow automatically move electrons from a chip's outgoing wires to another chip's incoming wires, so do VMBus instances provide the same for VM nodes. This is more than just a convenience, if VMBus did not exist, then each processor would need added functionality to forward such information or be faced with job data becoming stuck in outgoing queues[17].

The reader may have noticed the presence of the methods startListeningTo, stopListeningTo, startSpeakingTo, and stopSpeakingTo in VMBus. These methods allow the incoming job queue (i.e., startListeningTo and stopListeningTo) to become attached to another node's outgoing job queue, and conversely, the outgoing job queue (i.e., startSpeakingTo and stopSpeakingTo) to become attached to an incoming job queue. In creating such attachments, a dynamic state machine is formed. In general such a machine is a graph and no restrictions are placed on how the nodes are connected. However, if the kernel user does not want introduce a new Queue type that is a hybrid of VM and VMBus, then it is essential to have an instance of VMBus to connect any two VM instances. The kernel supports dynamic invocations of all VMBus methods in order to permit dynamic machine construction and evolution. Additionally, every VM permits the dynamic installation of Processors (i.e., Installer, see page 92) and dynamic processor control (i.e., Controller, see page 90). Each VM and VMBus has an Installer and

---

[17]Some may argue that VMBus' functionality could or should be implemented within a VM node. This was not done because it was decided to have each Processor perform a "single" task with the least amount of, possibly unused, state information as possible. Not only did this simplify the definition of all Processors, but, it also allowed different styles of internode broadcasts to be easily implemented (e.g., SynchronousDispatcher and AsynchronousDispatcher on pages 96 and 88 respectively) and changed at run-time.

Controller accessible by the methods getProcessorInstaller and getController respectively. Much of the functionality of both VM and VMBus nodes rely heavily on the ProducerConsumerMonitor class.

ProducerConsumerMonitor is a Queue-derived class that provides a thread-safe solution to *both* the finite and infinite producer-consumer problems permitting both synchronous and asynchronous queuing behaviour [BENAR90, STALL92][18]. Alone, it represents about two weeks of work designing its interface and testing it in a heavily multithreaded environment. Its final interface design is that of an abstract queue (i.e., Queue) and serves to hide all of Java's multithreading monitor mechanisms from a user of this kernel. This is important for two reasons: (i) it greatly simplifies the coding of inter-Thread communication when using large numbers of threads and (ii) to hide the Java virtual machine's monitor instructions (e.g., monitorenter and monitorexit, see [MEYER97]) and related keywords (i.e., synchronized). Sadly, Java does not use C. A. R. Hoare's communicating sequential processes (CSP) for process synchronization (i.e., [HOARE78]), instead it uses an earlier paper by Hoare (i.e., [HOARE74]) [BUYYA99b]. The ProducerConsumerMonitor class eliminates the need to use monitor related keywords within code blocks in order to facilitate code development and possible future development of a CSP framework. To support queuing behaviours other than the traditional first-in, first-out (i.e., FIFO) ordering, ProducerConsumerMonitor accepts any Queue instance *via* its constructor. In the thesis software implementation, a traditional FIFO queue (i.e., VectorAsQueue, see page 83) is used throughout. Surprisingly, the author wasn't able to find an object-oriented solution to the infinite and finite buffer producer-consumer problems in textbooks or the literature. Thus, it is indeed possible that this is the first public mention

---

[18]The reader will note the presence of isFull and isEmpty styled methods in Enqueue, Dequeue, and Queue (see pages 79, 78, and 83 respectively).

of such a solution.

Despite such useful functionality found in these kernel classes, the kernel performs nothing by itself. For the kernel to be useful, it is necessary to write programs which utilize such. Two such uses are detailed in the next section.

5.3. The Demonstration Examples

For reasons of development time and code complexity, the author decided to adopt a "keep it simple" philosophy with regard to the actual computation being performed with any implemented demonstration of the kernel. This is acceptable as the purpose of the demonstration is *not* to demonstrate a specific model of computation, rather, it is to demonstrate that the partially implemented kernel-architecture (i.e., the Job Metamodel) can actually be implemented and work properly. The operation of adding integers was chosen since addition is one of the simplest operations to write code for, run sequentially or in parallel, and verify the correctness of the computed result. Thus, the two demonstration programs were defined to compute the sum of a randomly generated array of positive `java.lang.Integers`. These integers were summed using the `java.math.BigInteger` class to properly handle large numbers.

Since Java renders deployment across the World Wide Web easy *via* `java.applet.Applet` with most commonly used web browsers, no software was needed to install the system on a user's machine as his/her web browser would do that when an appropriate page was downloaded. Therefore, the system needed only be set up on an appropriate set of servers so that the user could generate job requests and receive their results. The first demonstration runs entirely on the user's computer, intentionally avoiding nearly all network problems if machines are

down, etc. The second program runs across a set of computers connected across a network (i.e., Ethernet, ATM). Since network fault tolerance has not been implemented the second program is very sensitive to network failures and timeouts, hence, the reason for the deployment of the first program. Appendix C (see page 134) shows a variety of screen captures of the demonstration programs discussed in the next section.

5.3.1. The Single Node Demonstration

The single node demonstration consists of creating a virtual machine with seven (7) nodes[19] as is illustrated in Figure 9 below.

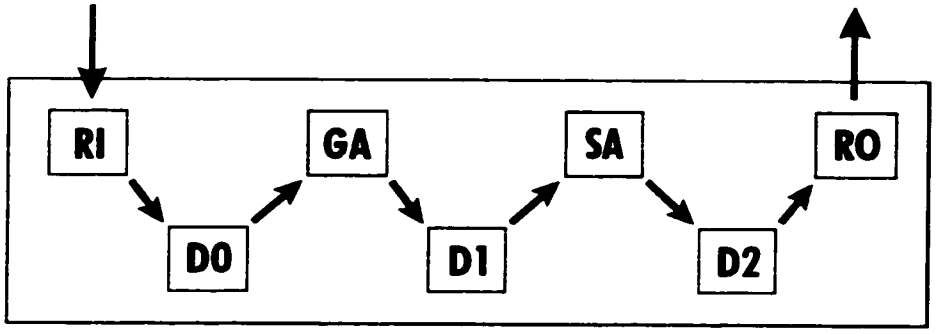

Figure 9. Single node demonstration program model.

In the figure, each arrowhead represents the act of enqueuing while each arrow tail represents the act of dequeuing. The machine nodes labelled, **RI**, **GA**, **SA**, and **RO**, represent `VM` nodes using a `ProcessorForAlgorithm` processor. These nodes are respectively coupled with `Algorithm_RenderInput`, `Algorithm_GenerateArray`, `Algorithm_SumArrayOnClient`, and `Algorithm_RenderOutput` found in Appendix B (see pages 122, 117, 126, and 123). The remaining nodes, **DO**, **D1**, and **D2** are `VMBus` instances. The grey box represents the created machine with the necessary incoming and outgoing job queue arrows effectively coming from the human user.

---

[19]This count excludes the necessary `VMBus` starting state to enqueue the data into **RI**.

RI obtains its input when the user clicks on the **Invoke Task...** button from the main applet screen as illustrated in Figure 11 on page 134. When invoked, **RI** brings up a window allowing the user to choose the random number seed used to generate the random numbers and the number of numbers that he/she wishes to sum. Practical limits have been placed on the range of these numbers. When these limits are exceeded error messages appear at the bottom of the window. Upon clicking **Ok**, those two numbers are sent to **DO** and are then forwarded to **GA**. If **Cancel** is chosen, then nothing is output from **RI** and that line of execution ceases. Otherwise, **GA** then generates an array of random `java.lang.Integers` of the appropriate length and outputs that to **D1** which forwards the information to **SA**. **SA** then sums the array and outputs it as a `java.math.BigInteger` instance to **D2** which forwards it to **RO**. Upon receiving the sum, **RO** displays it in a window (see Figure 14 on page 137) visible to the user. Miscellaneous debugging and timing information is also output in that window (see Figures 15 and 16 on pages 138 and 138 respectively). After the data are output to the window, the execution of the job has been completed.

This machine is initially created when the user downloads the applet, prior to the user clicking the **Invoke Task...** button. The code to actually create this machine once the Algorithm-derived classes for **RI**, **GA**, **SA**, and **RO** are written was very simple and can be found on page 113 in the constructor of `ComputeSumOfRandomIntegerArray_1CPU`. The simplicity of that code is testament to the simplicity of the design of `VM` and `VMBus`.

5.3.2. The Two Node Demonstration

The two node demonstration is nearly identical to the single node demonstration except that it (i) has more states and (ii) its states are spread across separate computers. However, this

does not affect the simplicity of setting up the machine, provided the proper input and output (e.g., network socket I/O) devices are represented as Virtual Machine Node implementations. The two-node demonstration consists of thirteen (13) nodes[20] presented in Figure 10 below.

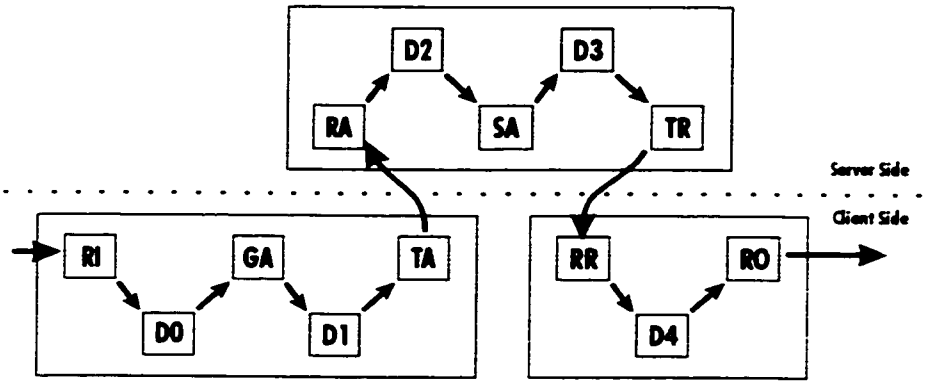

Figure 10. Two node demonstration program model.

In this figure **D0**, **D1**, **D2**, **D3**, and **D4** are `VMBus` nodes while **RI**, **GA**, **TA**, **RA**, **SA**, **TR**, **RR**, and **RO** are `VM` nodes. The `VM` nodes correspond to a variety of `Algorithm`-derived classes: **RI** with `Algorithm_RenderInput` (see page 122), **GA** with `Algorithm_GenerateArray` (see page 117), **TA** with `Algorithm_TransmitArrayToBroker` (see page 127), **RA** with `Algorithm_ReceiveArrayFromClient` (see page 119), **SA** with `Algorithm_SumArray` (see page 124), **TR** with `Algorithm_TransmitResultToClient` (see page 129), **RR** with `Algorithm_ReceiveResultFromBroker` (see page 120), and **RO** with `Algorithm_RenderOutput` (see page 123). The purpose of each state is indicated in its class name after "`Algorithm_`". Additionally, **RI**, **GA**, and **RO** are identical to the nodes of the same label in the single node demonstration earlier. Similarly, **SA** is nearly identical to the node labelled as **SA** in the single node example; —instead of outputting debugging and timing information to a window on the user's screen, it outputs the information to the standard output device on the server it is running. Of the remaining nodes, **TA** and **TR** are output devices. These devices

---

[20]As in the single node example, this count excludes the necessary `VMBus` starting state to enqueue the data into **RI**.

use Java's built-in serialization mechanism to output, *via* a TCP network socket, the array and summed result to the broker and client respectively. Conversely, **RA** and **RR** are input devices receiving the outputs from **TA** and **TR** respectively. Therefore, this machine is identical to the Figure 9 earlier except that it inserts **TA** and **RA** between **GA** and **SA**, and, **TR** and **RR** between **SA** and **RO**. This is necessary to accomplish the required network socket communication to run this machine across two computers. From the user's perspective there is no difference between this and the single node examples except that the "Monitor" window omits the detail from the output to the standard console on the server machine (see Figure 13 on page 136).

As with the single node example, this machine is initially created when the user downloads the applet, prior to the user clicking the **Invoke Task...** button. The code to actually create this machine can be found on page 115 in the constructor and `main()` function descriptions found with the class `ComputeSumOfRandomIntegerArray_2CPUs`. The code there is almost as simple as found in the single-CPU demonstration.

5.4. General Discussion of the Kernel Implementation & Examples

The two demonstration examples discussed above clearly show that it is indeed possible to implement the required kernel insofar as discussed in this thesis. Each implemented component of the Job Metamodel kernel was itself an autonomous Virtual Machine Node implementation with the ability to be utilized in a variety of software environments within a Java virtual machine. This was possible by the encapsulation of all internal Virtual Machine Node components, except the external queue representing the broker interface. This provided a *loose coupling* between the internal machinery of a node and that of the broker interface which the user and internal machinery interact with. Within the realm of object-oriented computing, such loose couplings are desired and have many beneficial consequences

[PFLEE98].

The presented examples were designed to simply illustrate that a working kernel could be implemented on a single computer and across a computer network. Hence, the examples do not attempt to fully realize the true power and flexibility of the kernel's design. Since each VM is a Queue, it is possible to install entire VM instances in place of the incoming and outgoing job queues at the point of VM instance construction as was shown in Figure 8 on page 49. Such can be used to hide data flows to and from VM instances across machine address boundaries. This is best illustrated with an example. Consider again Figure 9 (see page 60). The final state machine illustrated computes the sum of an array of integers within a single address space (i.e., a single CPU computer). However, abstractly, the same figure could be argued to represent the computation in the two node demonstration (i.e., two networked computers). For this to hold true, the job data flow through the edge (i.e., arrow) from **GA** to **SA** has to instead perform the *same* transition through **TA** and **RA** in Figure 10 (see page 62). In Figure 9, such a transition would be argued to be "hidden" inside **GA**'s outgoing job queue and **SA**'s incoming job queue. Thus, an evolutionary compiler monitoring the status of a dynamically changing distributed networking environment could easily modify that transition in Figure 9 to that of Figure 10 by installing **TA** as **GA**'s outgoing job queue and **RA** as **SA**'s incoming job queue. So long as such changes have the same fault tolerant properties and *effective* job semantics before and after such a transformation, the transition will be guaranteed to perform properly in each case. Thus, if the transition from **SA** to **RO** is modified by similar logic through the installation of **TR** and **RR** as queues, Figure 9 can fully represent Figure 10 and *vice*

versa[21]. Thus, the design of the queuing mechanisms in the Virtual Machine Node Model allow for the ***dynamic***, not just ***static***, transformation of the ***deployment*** of virtual machine nodes in some appropriately relevant address space (e.g., network, machine address boundaries). To do such transformations automatically and meaningfully, requires a much larger Job Metamodel kernel that is fully aware of the state automaton represented by the machine(s) being modelled. This kernel, in fact, would have to be a compiler with enough machine metainformation and general communication infrastructure deployed across all participating machines to realize this goal. This *is* precisely the long-term aim of the author's personal and academic research into metamodelling. Achieving the first part of this, a execution framework for the simulation of modelling, *was* the primary aim of this thesis.

---

[21]For simplicity, it is assumed in this discussion that there is no debug and timing related monitoring information used or produced by **RI**, **GA**, **TA**, **RA**, **SA**, **TR**, **RR**, **RO**, and any of the dispatching nodes. If such monitoring information appeared it would be represented explicitly by VM node instances for such purposes. The monitor information in this thesis was to merely provide feedback to the researcher and user of the kernel's design and functionality and *not* to be part of any actual simulated machine execution framework.

CHAPTER 6.
Conclusions

This thesis asked, "How can one design and implement a computing system that minimizes the work in designing, implementing, and maintaining any computing system including itself?" In so doing, it considered academic, experimental, and personal information which could not possibly be exhaustively documented in any thesis. It was explained that to strive towards this goal was to attempt to create a computing system capable of metamodelling. Those considerations lead to the presentation the Modelling Cycle and Job Metamodels. These two Metamodels were argued as an important start to begin addressing what must be the architecture and kernel of a future computing system that achieves this thesis' idealistic goals. Thus, the Modelling Cycle and Job Metamodels along with the partial kernel implementation of the latter are the achievements of this thesis. Such achievements and some additional comments are detailed below.

6.1. Thesis Achievements

The first achievement of this thesis was the Modelling Cycle Metamodel. This model was proposed to better understand, research and recognize aspects of the life history, survivorship, and evolution of models and their implementations. While not directly implemented in the modelling kernel and its demonstration within this thesis, it was used to aid Job Metamodel's design and future research that must account for the evolution of models in a computing system.

The second and most important achievement of this thesis was the Job Metamodel. The Job Metamodel characterized what a computing system, job, user, and broker interface are and

how they interact with each other *via* the broker interface. To provide for an eventual complete, formal Job Metamodel, the Virtual Machine Node Model presented a machine architecture for any computing system suited to the task of (meta)modelling and that of designing, implementing, and maintaining computing systems. Further, the Virtual Machine Node Model distinguishes itself from Turing machines by only considering the interfaces by which computations are performed rather than how they are done. The question of how computations should and ought to be performed was intentionally left undefined in the model itself. This allows for the potential of larger problem sets to be entertained by computing systems than those based on the universal Turing machine. Last, the Virtual Machine Node Model can be used to create systems representing all of Flynn's architectural models (i.e., SISD, SIMD, MISD, MIMD) and, thus, the model has the ability to *at least* represent those systems.

Finally, an implementation of the Virtual Machine Node Model and various aspects of the Job Metamodel, complete with two working examples, were implemented to execute within the Java virtual machine. Besides lending support that the presented model can be practically created and utilized, the software provided a simple representation of the proposed computing system architecture running across one and two node systems. This representation was able to use the interface of an abstract `Queue` coupled with some additional functions to allow for the construction of the state machine. Further, the external, incoming, and outgoing job queues of the Virtual Machine Node Model were easily implemented and provide the necessary framework to encapsulate and permit the dynamic transformation of computing system behaviours, deployment, and policy enforcement. The latter addressed an essential requirement in changing cluster computing environments.

6.2. Some Comments and Directions

The Modelling Cycle and Job Metamodels *defined* the rationale on a set of *abilities* that a metamodelling system ought to be able to represent and accomplish. Most importantly, these metamodels allowed job computations to be performed, modelled, and evolved over time. Whereas static implementations would allow for greater efficiency, dynamic ones would allow for an increase in model survivorship in environments of change. Examples of relatively static environments include today's non-networked computers, their applications, and their networks. However, there is a greater need for virtual computers and networks that work with various types of physical computer and network setups. If one considers the enforcement of institutional, corporate, and individual copyright, trademark, patent, and other so-called intellectual property policies of machines on the Internet, as well as nested and overlapping subsets thereof, one soon realizes that these environments are *very* dynamic, driven by constant asynchronous change amongst various *collaborative* entities.

Further, many of these collaborating entities are researchers that are responsible for developing and testing *models* as *hypotheses* against other models in a formal exercise called *natural philosophy*; —*science*. In the end, collaboration is what is important to metacomputing, science, and society; —for collaboration is the *coupling* of some set of *interfaces* (e.g., human beings talking to one another, intelligent software agents communicating, etc.) used as a *means* to an end. Given that this thesis' definition of computation is nothing more than a *means* to perform some operation, modelling and metamodelling are collaborative exercises aimed at providing some end through the use of various existing, defined, and/or implemented collaborative agents.

Besides providing the Modelling Cycle and Job Metamodels, this thesis provided an implementation of the Virtual Machine Node Model, a key component of the Job Metamodel. This implementation was a dynamically created virtual machine that satisfied the idealistic goals as dictated by the Job Metamodel, and, established a basis by which further research and development can occur.

The author intends to pursue further research and development of this thesis and other, more general, metacomputing and metamodelling goals. One product will likely be the creation of virtual code deployment machines with built-in compilers and simulators. Such machines will be coupled with metamodelling frameworks (e.g., *via* metaclasses) that are "self-aware" of all of their contained models, to research the capability of models and implementations to automatically adapt to selection pressures of various types of computing environments. In so doing, metamodels of policy enforcement should be able to be coupled with compiler technology to work together with the Modelling Cycle, Job, and other metamodels efficiently and automatically in any such system. While such could easily drive several large projects, all projects are accomplished one step and one day at a time. This thesis was one such step.

APPENDIX A.
Kernel Code

A.1. Overview

This appendix contains all of the `public` and `protected` `classes` and `interfaces` for all of this thesis' kernel code. The code is presented hierarchically first by `package` and then by `class/interface` in alphabetical order. All `public` and `protected` methods have their prototypes given and have their functionality described. All of the code uses valid Java syntax except code listed for Java classes, since the method definition is omitted. However, in all other respects, all Java classes listed herein are syntactically correct.

A.2. Package: `preneypaul.msc.libs.dp`

A.2.1. Overview

This package contains classes that qualify as design patterns and design pattern wrappers.

A.2.2. Classes Hierarchy

```
(java.lang.Object)
    (java.lang.Throwable)
        (java.lang.Exception)
            BasicException
```

A.2.3. Classes

`BasicException`

A.2.4. Interfaces

`Iterator`

A.2.5. Class: `BasicException`

A.2.5.1. Overview

A `BasicException` was designed to represent/hold any type of exception that is capable of being thrown in Java. Since the kernel code was designed to run third-party client code, there has to be a way to capture any exception that is thrown and to allow that to be properly re-thrown in such a manner that it is handled as a `java.langException`-derived object. Specifically, a wrapper class is needed for all classes derived from `java.lang.Throwable`. Thus, this pattern permits proper error handling and reporting of *all* errors that occur in third-party code executed within the kernel.

A.2.5.2. Definition

```
public class BasicException extends Exception
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public BasicException();
    public BasicException(Exception anException);
    public BasicException(String aString);
    public BasicException(Throwable aThrowable);
    public Exception getInvokingException();
    public Throwable getInvokingThrowable();
    public boolean isExceptionInvoked();
    public boolean isThrowableInvoked();
}
```

A.2.5.3. Methods

A.2.5.3.1. `BasicException()`

Equivalent to `java.lang.Exception`'s `Exception()` constructor.

A.2.5.3.2. `BasicException(Exception anException)`

Equivalent to invoking `java.lang.Exception`'s `Exception()` constructor except that it also stores a reference to `anException`; i.e., `isExceptionInvoked()` and `isThrowableInvoked()` will both return `true`.

A.2.5.3.3. BasicException(String aString)

Equivalent to java.lang.Exception's Exception(String aString) constructor.

A.2.5.3.4. BasicException(Throwable aThrowable)

Equivalent to java.lang.Exception's Exception() constructor except that it also stores a reference to aThrowable; i.e., isThrowableInvoked() will return true.

A.2.5.3.5. Exception getInvokingException()

If isExceptionInvoked() returns true, then the java.lang.Exception-derived object passed to this object's constructor will be returned. Otherwise, null is returned or a java.lang.ClassCastException is thrown.

A.2.5.3.6. Throwable getInvokingThrowable()

If isThrowableInvoked() returns true, then the java.lang.Throwable-derived object passed to this object's constructor will be returned. Otherwise, null is returned.

A.2.5.3.7. boolean isExceptionInvoked()

If this object was created by passing in a java.lang.Exception-derived object, then this method returns true; false otherwise.

A.2.5.3.8. boolean isThrowableInvoked()

If this object was created by passing in a java.lang.Throwable-derived object, then this method returns true; false otherwise.

A.2.6. Interface: `Iterator`

A.2.6.1. Overview

The `Iterator` pattern is an abstract representation of traversing through a sequence of objects in a well-defined order.

A.2.6.2. Definition

```
public interface Iterator
{
    static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    Object getItem();
    void goToFirst();
    void goToNext();
    boolean isDone();
}
```

A.2.6.3. Methods

A.2.6.3.1. `Object getItem()`

Returns the current object in the sequence if `isDone()` returns `false`; otherwise, `null` is returned.

A.2.6.3.2. `void goToFirst()`

Tells the `Iterator` to (re-)start iterating from the beginning of the sequence.

A.2.6.3.3. `void goToNext()`

Tells the `Iterator` to retrieve the next item in the sequence.

A.2.6.3.4. `boolean isDone()`

Returns `true` if there are no more elements in the sequence; `false` otherwise.

A.3. Package: `preneypaul.msc.libs.dp.impl`

A.3.1. Overview

This package consists of specific implementations of the `preneypaul.msc.libs.dp` package.

A.3.2. Classes Hierarchy

```
(java.lang.Object)
    IteratorForEmptyContainer
    IteratorForVector
```

A.3.3. Classes

```
IteratorForEmptyContainer
IteratorForVector
```

A.3.4. Class: IteratorForEmptyContainer

A.3.4.1. Overview

An IteratorForEmptyContainer represents a zero-length sequence. This object was created to avoid having to have an zero-length sequence object, typically derived from preneypaul.msc.libs.ds.Container, defined just to create an Iterator which would never iterate through any elements since there were none. Thus, this object saves both time and RAM resources.

A.3.4.2. Definition

```
import preneypaul.msc.libs.dp.Iterator;
final public class IteratorForEmptyContainer implements Iterator
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public IteratorForEmptyContainer();
    public Object getItem();
    public void goToFirst();
    public void goToNext();
    public boolean isDone();
}
```

A.3.4.3. Methods

See the superclass for unlisted method descriptions.

A.3.4.3.1. IteratorForEmptyContainer()

Constructs an Iterator for a zero-length sequence; i.e., isDone() always returns true.

A.3.5. Class: `IteratorForVector`

A.3.5.1. Overview

While `java.util.Enumeration` exists to iterate through a Vector, it was preferred to have an `Iterator`-derived iterator pattern object used universally throughout the kernel code for consistency. The observant reader will note that `Iterator` differs from `java.util.Enumeration` in how one tells the iterator object to iterate through the sequence.

A.3.5.2. Definition

```
import preneypaul.msc.libs.dp.Iterator;
import java.util.Vector;
final public class IteratorForVector implements Iterator
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public IteratorForVector(Vector aVector);
    public Object getItem();
    public void goToFirst();
    public void goToNext();
    public boolean isDone();
}
```

A.3.5.3. Methods

See the superclass for unlisted method descriptions.

A.3.5.3.1. `IteratorForVector(Vector aVector)`

Constructs an `Iterator` that will iterate through `aVector` from its lowest numbered index to its highest.

A.4. Package: `preneypaul.msc.libs.os`

A.4.1. Overview

This package was created to hold operating system (OS), i.e., kernel, relevant objects.

A.4.2. Classes Hierarchy

```
(java.lang.Object)
    (java.lang.Throwable)
```

```
(java.lang.Exception)
    (preneypaul.msc.libs.dp.BasicException)
        OSException
```

A.4.3. Classes

```
OSException
```

A.4.4. Class: OSException

A.4.4.1. Overview

Based on BasicException, this object serves as the primary base exception class for *all* of the kernel code.

A.4.4.2. Definition

```
import preneypaul.msc.libs.dp.BasicException;
public class OSException extends BasicException
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public OSException();
    public OSException(Exception anException);
    public OSException(String aString);
    public OSException(Throwable aThrowable);
}
```

A.4.4.3. Methods

See superclass for method descriptions.

A.5. Package: preneypaul.msc.libs.os.ds

A.5.1. Overview

This package contains all operating system, i.e., kernel-relevant, data structures. For power and flexibility, nearly all of the data structures are defined as abstract interfaces.

A.5.2. Classes Hierarchy

```
(java.lang.Object)
    ObjectHolder
```

A.5.3. Classes

```
ObjectHolder
```

A.5.4. Interfaces

```
Container
Dequeue
Enqueue
InteratableContainer
Queue
```

A.5.5. Interface: `Container`

A `Container` holds a collection of objects. As this is an abstract interface, the representation of exactly what a `Container` is, must be defined by derived classes.

A.5.5.1. Definition

```
public interface Container
{
    static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    boolean isEmpty();
    boolean isFull();
}
```

A.5.5.2. Methods

A.5.5.2.1. `boolean isEmpty()`

Returns `true` if the `Container` is empty, i.e., the `Container` is presently holding no (contained) objects; `false` otherwise.

A.5.5.2.2. `boolean isFull()`

Returns `true` if the `Container` is full, i.e., the `Container` is presently holding the maximum number of (contained) objects it can; `false` otherwise.

A.5.6. Interface: Dequeue

A.5.6.1. Overview

A Dequeue represents a half-queue, specifically, the half that permits the removal of objects from some queue. All derived instances of this interface *must* be thread-safe and *must* obey the defined blocking behaviours. This class is typically used with the Enqueue and Queue interfaces also in this package.

A.5.6.2. Definition

```
public interface Dequeue extends Container
{
    static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    Object dequeue();
    boolean isDequeueEmpty();
    boolean isDequeueFull();
    boolean tryDequeue(ObjectHolder anObject);
}
```

A.5.6.3. Methods

A.5.6.3.1. Object dequeue()

Returns the next object to be removed from the Dequeue, i.e., some Queue. If the queue is empty, i.e., isDequeueEmpty() returns true, then the method *blocks* until some object can be removed from the said Queue. If such blocking behaviour is not desired, then the tryDequeue method should be used instead.

A.5.6.3.2. boolean isDequeueEmpty()

Returns true if no elements can be removed from the Dequeue; false otherwise. **Note:** Even if this method returns true, in a multithreaded environment, one is *not* guaranteed that there will be an element to remove from the queue after calling this method.

A.5.6.3.3. boolean isDequeueFull()

Returns true if the Dequeue can hold no more elements; false otherwise. This method has limited utility except when it is used as part of the Queue interface presented later in this package.

A.5.6.3.4. boolean tryDequeue(ObjectHolder anObject)

Returns true if an element was removed from the Dequeue; false otherwise. If true is returned, anObject holds the object removed from the queue, i.e., anObject.isDefined() will return true. If false is returned, anObject.isDefined() will return false. Unlike dequeue(), this method will *never* block.

A.5.7. Interface: Enqueue

A.5.7.1. Overview

An Enqueue represents a half-queue, specifically, the half that permits the addition of objects to some queue. All derived instances of this interface *must* be thread-safe and *must* obey the defined blocking behaviours. This class is typically used with the Dequeue and Queue interfaces also in this package.

A.5.7.2. Definition

```
public interface Enqueue extends Container
{
    static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    void enqueue(Object anObject);
    boolean isEnqueueEmpty();
    boolean isEnqueueFull();
    boolean tryEnqueue(Object anObject);
}
```

A.5.7.3. Methods

A.5.7.3.1. `void enqueue(Object anObject)`

Adds `anObject` to the `Enqueue`, i.e., some `Queue`. If the queue is full, i.e., `isEnqueueFull()` returns `true`, then the method *blocks* until the object can be added to the said `Queue`. If such blocking behaviour is not desired then the `tryEnqueue` method should be used instead.

A.5.7.3.2. `boolean isEnqueueEmpty()`

Returns `true` if there are no elements that can be removed from the `Enqueue`; `false` otherwise. This method has limited utility except when it is used as part of the `Queue` interface presented later in this `package`.

A.5.7.3.3. `boolean isEnqueueFull()`

Returns `true` if the `Enqueue` can hold no more elements; `false` otherwise. **Note:** Even if this method returns `false`, in a multithreaded environment, one is *not* guaranteed that an element can be added to the `Enqueue` after calling this method.

A.5.7.3.4. `boolean tryEnqueue(Object anObject)`

Returns `true` if `anObject` was added to the `Enqueue`; `false` otherwise. Unlike `enqueue()`, this method will *never* block.

A.5.8. Interface: `IteratableContainer`

A.5.8.1. Overview

Since it may be possible to iterate, in some standard way, through many different types of `Containers`, this interface, `IteratableContainer`, defines such. In this way, there will exist a standard way to obtain an `Iterator` for a `Container`.

A.5.8.2. Definition

```
import preneypaul.msc.libs.dp.Iterator;
public interface IteratableContainer
{
    static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    Iterator iterate();
}
```

A.5.8.3. Methods

A.5.8.3.1. `Iterator iterate()`

Returns an `Iterator` capable of iterating through some sequence of elements in some standard way for that object instance. All implementations of this method *must* be thread-safe and the `IteratableContainer` must *not* change while iterating through its sequence. That is, the sequence that the returned `Iterator` object iterates through must be *invariant* over the returned `Iterator` object's lifetime.

A.5.9. Class: `ObjectHolder`

A.5.9.1. Overview

Although Java supports passing arguments to methods by reference, such arguments cannot be used to pass back the `null` value. `ObjectHolder` was defined to allow passing via arguments to methods arbitrary values back to the method caller. Additionally, the state of the object being passed back is tracked as well; i.e., if the `ObjectHolder` instance holds any valid `Object` value, including `null`, then `isDefined()` will return `true`; otherwise, `isDefined()` will return `false`.

A.5.9.2. Definition

```
public final class ObjectHolder
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public ObjectHolder();
    public ObjectHolder(final Object anObject);
    public final synchronized Object getObject();
    public boolean isDefined();
```

```
    public final synchronized void setObject(Object anObject);
    public final synchronized void undefine();
}
```

A.5.9.3. Methods

A.5.9.3.1. `ObjectHolder()`

Creates an `ObjectHolder` instance that sets the status returned by `isDefined()` to be `false`.

A.5.9.3.2. `ObjectHolder(final Object anObject)`

Creates an `ObjectHolder` instance that holds `anObject` and sets the status returned by `isDefined()` to be `true`.

A.5.9.3.3. `Object getObject()`

Returns the object being held if the status returned by `isDefined()` is `true`; otherwise `null` is returned.

A.5.9.3.4. `boolean isDefined()`

Returns `true` if an object is being held; `false` otherwise.

A.5.9.3.5. `void setObject(Object anObject)`

Tells the `ObjectHolder` instance to hold `anObject` and set the status returned by `isDefined()` to be `true`.

A.5.9.3.6. `void undefine()`

Tells the `ObjectHolder` instance to release any object it may be holding and set the status returned by `isDefined()` to be `false`.

A.5.10. Interface: Queue

A.5.10.1. Overview

A Queue represents a full-queue, specifically, the functionality of an Enqueue and a Dequeue combined into a single object. Enqueue and Dequeue are interfaces also defined in this package.

A.5.10.2. Definition

```
public interface Queue extends Enqueue, Dequeue
{
    static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";
}
```

A.5.10.3. Methods

See the superclasses for method descriptions.

A.6. Package: preneypaul.msc.libs.os.ds.impl

A.6.1. Overview

This package contains classes that are implementations of classes/interfaces defined in the preneypaul.msc.libs.os.ds package.

A.6.2. Classes Hierarchy

```
(java.lang.Object)
    VectorAsQueue
```

A.6.3. Classes

VectorAsQueue

A.6.4. Class: VectorAsQueue

A.6.4.1. Overview

VectorAsQueue is an implementation of the Queue and IteratableContainer interfaces that uses java.util.Vector. For its efficiency and simplicity, the kernel uses this class for most of its

queuing support.

A.6.4.2. Definition

```
import preneypaul.msc.libs.dp.Iterator;
import preneypaul.msc.libs.os.ds.ObjectHolder;
public class VectorAsQueue implements Queue, IteratableContainer
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public VectorAsQueue();
    public synchronized Object dequeue();
    public synchronized void enqueue(Object anObject);
    public boolean isDequeueEmpty();
    public boolean isDequeueFull();
    public boolean isEmpty();
    public boolean isEnqueueEmpty();
    public boolean isEnqueueFull();
    public boolean isFull();
    public synchronized Iterator iterate();
    public synchronized boolean tryDequeue(ObjectHolder anObject);
    public synchronized boolean tryEnqueue(Object anObject);
}
```

A.6.4.3. Methods

See the superclasses for unlisted method descriptions.

A.6.4.3.1. `VectorAsQueue()`

Constructs a `VectorAsQueue` object with its `Queue` initially empty.

A.7. Package: `preneypaul.msc.libs.os.monitors`

A.7.1. Overview

This package contains a series of specialized "monitor" classes essential for synchronizing concurrent threads/processes. All multithreading synchronization behaviours in the kernel are controlled with objects from this package.

Note: Originally, this package contained several "monitor" classes, including various types of semaphores and older versions of solutions of the well known producer-consumer problem. All of those classes were made obsolete and were removed with the (only) remaining

class, ProducerConsumerMonitor, as its blocking behaviours are implicit and it is guaranteed to avoid resource deadlock.

A.7.2. Classes Hierarchy

```
(java.lang.Object)
    ProducerConsumerMonitor
```

A.7.3. Classes

```
ProducerConsumerMonitor
```

A.7.4. Class: ProducerConsumerMonitor

A.7.4.1. Overview

The ProducerConsumerMonitor class implements both the finite- and infinite-buffer solutions to the producer consumer problem [BENAR90, STALL92]. This class is the single most important class in the kernel for it is the *only* one that is used to synchronize threads. Since this class *guarantees* resource queue deadlock will *not* occur *between* concurrently executing program sections, it was a lot easier to ensure that code using this class was also correct. The astute reader will note that the reason why this solution is adequate for the infinite-buffer producer-consumer problem is that it operates on *abstract* Queues; —i.e., there is no "real", hard-coded notion of when isEmpty() and isFull() actually occur, nor, is there a "real", hard-coded notion of how a Queue is represented on any given system.

Note: The class design for the ProducerConsumerMonitor is the product of several weeks of code design and testing. The final form, presented here, is not only elegant and correct, but also, permits both synchronous and asynchronous queuing behaviours. Much of the code design and testing of this class effected the design of the Container, Dequeue, Enqueue, and Queue classes. I know of no published, object-oriented solution to the producer-consumer problem that gives

such flexibility.

A.7.4.2. Definition

```
import preneypaul.msc.libs.dp.Iterator;
import preneypaul.msc.libs.os.ds.ObjectHolder;
public final class ProducerConsumerMonitor implements Queue
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public ProducerConsumerMonitor(Queue aQueue);
    public Object dequeue();
    public void enqueue(final Object anObject);
    public boolean isDequeueEmpty();
    public boolean isDequeueFull();
    public boolean isEmpty();
    public boolean isEnqueueEmpty();
    public boolean isEnqueueFull();
    public boolean isFull();
    public boolean tryDequeue(ObjectHolder anObject);
    public boolean tryEnqueue(final Object anObject);
}
```

A.7.4.3. Methods

See the superclass for unlisted method descriptions.

A.7.4.3.1. `ProducerConsumerMonitor(Queue aQueue)`

Constructs a `ProducerConsumerMonitor` instance that acts upon the `Queue`, `aQueue`.

A.8. Package: `preneypaul.msc.libs.os.vm`

A.8.1. Overview

This package contains all of the essential classes for the kernel code.

A.8.2. Classes Hierarchy

```
(java.lang.Object)
    AsynchronousDispatcher
    CallReturnSpace
    Processor
        ProcessorForAlgorithm
        ProcessorForDispatcher
    SynchronousDispatcher
    (java.lang.Throwable)
        (java.lang.Exception)
            (preneypaul.msc.libs.dp.BasicException)
                (preneypaul.msc.libs.os.OSException)
                    VMException
```

VM
VMBus

A.8.3. Classes

AsynchronousDispatcher
CallReturnSpace
Processor
ProcessorForAlgorithm
ProcessorForDispatcher
SynchronousDispatcher
VM
VMBus
VMException

A.8.4. Interfaces

Algorithm
Controller
Dispatcher
Installer

A.8.5. Interface: Algorithm

A.8.5.1. Overview

The Algorithm interface represents an abstract algorithm, function, or procedure. Every Algorithm may accept some input and may produce some output. The primary purpose of this interface is to remove third-party code dependency on the Processor class also defined in this package. In this way, a third-party can write Algorithms *independent* of the type Processor that they are to run on.

A.8.5.2. Definition

```
import preneypaul.msc.libs.os.ds.ObjectHolder;
public interface Algorithm
{
    static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    void algorithm(Object anInput, ObjectHolder anOutput);
}
```

A.8.5.3. Methods

A.8.5.3.1. `void algorithm(Object anInput, ObjectHolder anOutput)`

This method will execute some implementation-specified algorithm given `anInput` and will (optionally) produce some output to be held in `anOutput`. Absolutely no exceptions should be thrown from this method. If any exceptions are thrown, they will be noted by the `Processor` executing this `Algorithm` and will subsequently be ignored.

A.8.6. Class: `AsynchronousDispatcher`

A.8.6.1. Overview

An `AsynchronousDispatcher` asynchronously broadcasts objects to a set of "listening" objects. Analogously, if one was talking about computer hardware, this class embodies the act of broadcasting data asynchronously across the computer bus to some number of hardware processing units. This class is considered a function object.

A.8.6.2. Definition

```
import preneypaul.msc.libs.dp.*;
public final class AsynchronousDispatcher implements Dispatcher
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public AsynchronousDispatcher();
    public final void dispatch(final Object anObject, final Iterator aListeners);
    public String toString();
}
```

A.8.6.3. Methods

A.8.6.3.1. `AsynchronousDispatcher()`

Constructs an instance of an `AsynchronousDispatcher`.

A.8.6.3.2. `void dispatch(final Object anObject, final Iterator aListeners)`

Dispatches `anObject` to the listeners defined in the sequence, `aListeners`. The sequence must be

of type VM, a class, that is also defined in this package.

A.8.6.3.3. public String toString()

Returns the string "AsynchronousDispatcher". Used for logging purposes.

A.8.7. Class: CallReturnSpace

A.8.7.1. Overview

CallReturnSpace is *experimental* class meant to conform to part of the TUPLE type definition in the *Tutorial D* specification defined in [DATE98]. In this thesis, this class serves as a mechanism to supply an *unordered* set of *typed* arguments as input to and output from the Algorithm class' algorithm method and as the key *message-passing* type transmitted amongst Processors. Specifically, a CallReturnSpace defines a one-to-one associative dictionary function: java.lang.Class × java.lang.String → java.lang.Object. Thus, this dictionary allows a given object type to be given a java.lang.String name to further identify that object.

A.8.7.2. Definition

```
public class CallReturnSpace
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public CallReturnSpace();
    public Object get(Class aClass, String aName);
    public void put(Class aClass, String aName, Object anObject);
}
```

A.8.7.3. Methods

A.8.7.3.1. CallReturnSpace()

Constructs an instance of CallReturnSpace.

A.8.7.3.2. Object get(Class aClass, String aName)

Returns the object associated with a specific type, aClass, with the name, aName. If there is no

object associated with `aClass` and `aName`, then `null` is returned.

A.8.7.3.3. `void put(Class aClass, String aName, Object anObject)`

Stores `anObject` associated with the type, `aClass`, and the name, `aName`.

A.8.8. Interface: `Controller`

A.8.8.1. Overview

A `Controller` defines an abstract interface that allows run-time process/thread control of a `VM` and its internal `Processor`, if any. This places, as is proper, control of a `VM` to outside the `VM` itself; e.g., a human controls a computer by starting it, turning it on, stopping it, turning it off, etc. tasks which the computer machinery cannot itself accomplish without "outside" help.

A.8.8.2. Definition

```
public interface Controller
{
    static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    VM getControlledVM();
    boolean isRunning();
    boolean isSuspended();
    void resume();
    void start();
    void startOrResume();
    void stop();
    void suspend();
}
```

A.8.8.3. Methods

A.8.8.3.1. `VM getControlledVM()`

Returns the `VM` that is being controlled by the `Controller` instance.

A.8.8.3.2. `boolean isRunning()`

Returns `true` if the associated `VM`'s `Processor` machinery is running; `false` otherwise.

A.8.8.3.3. `boolean isSuspended()`

Returns `true` if the associated `VM`'s `Processor` machinery is suspended; `false` otherwise.

A.8.8.3.4. `void resume()`

Resumes running the associated `VM`'s `Processor` machinery if it was previously suspended.

A.8.8.3.5. `void start()`

Starts running the associated `VM`'s `Processor` machinery only if is not already running.

A.8.8.3.6. `void startOrResume()`

Starts or resumes, as appropriate, running the associated `VM`'s `Processor` machinery if it was not already running or if it was previously suspended.

A.8.8.3.7. `void stop()`

Stops running the associated `VM`'s `Processor` machinery if it was already running.

A.8.8.3.8. `void suspend()`

Suspends running the associated `VM`'s `Processor` machinery if it was already running.

A.8.9. Interface: `Dispatcher`

A.8.9.1. Overview

A `Dispatcher` broadcasts objects to a set of "listening" objects. Analogously, if one was talking about computer hardware, this class embodies the act of broadcasting data across the computer bus to some number of hardware processing units. This class is considered a function object.

A.8.9.2. Definition

```
import preneypaul.msc.libs.dp.*;
public interface Dispatcher
{
    static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    void dispatch(Object anObject, Iterator aListeners) throws VMException;
}
```

A.8.9.3. Methods

A.8.9.3.1. `void dispatch(Object anObject, Iterator aListeners) throws VMException`

Dispatches `anObject` to the listeners defined in the sequence, `aListeners`.

A.8.10. Interface: `Installer`

A.8.10.1. Overview

An `Installer` defines an abstract interface that allows run-time installation and de-installation of `Processor` instances to specific `VM` instances. This places, as is proper, the installation of a `VM` component to be outside the `VM` itself; e.g. a human installing/removing chips from a motherboard is something that (normal) computer machinery cannot do itself without "outside" help.

Note: This class only permits a single object to be installed at any given time for a given `Installer` instance. However, derived implementations are free to define *what* they are installing. The installed object is called a "plugin".

A.8.10.2. Definition

```
public interface Installer
{
    static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    Object getPlugIn();
    boolean isPluggedIn();
    Object plugIn(Object anObject) throws VMException;
    Object unplug() throws VMException;
}
```

A.8.10.3. Methods

A.8.10.3.1. `Object getPlugIn()`

Returns the installed object; `null`, if no object is installed.

A.8.10.3.2. `boolean isPluggedIn()`

Returns `true` if an object is installed, `false` otherwise.

A.8.10.3.3. `Object plugIn(Object anObject) throws VMException`

Installs `anObject`, if possible. If `anObject` is successfully installed, then the method returns the previously installed object (it `unplugs` the previously installed object first). If there was no previously installed object, then null is returned. Otherwise, the installation was unsuccessful and a `VMException` is thrown. If the latter occurs, then the previously installed object remains installed.

A.8.10.3.4. `Object unplug() throws VMException`

Returns the installed object after de-installing it first.

A.8.11. Class: `Processor`

A.8.11.1. Overview

A `Processor` defines an abstract interface that defines a specific entity capable of doing some type of work. In this thesis, the work is that of running an `Algorithms` and `Dispatchers`. The reader should note that each processor, for it to be of use, must be installed, by an `Installer`, in some type of `VM` instance. Once installed, those `Processors` can be controlled through two mechanisms: the `VM`'s `Controller` object and/or the *data flow* through the machine.

A.8.11.2. Definition

```
import preneypaul.msc.libs.os.ds.*;
public abstract class Processor
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public Processor();
    public getVMWhereThisProcessorIsInstalled();
    public boolean isThisProcessorInstalled();
    public abstract void process(Dequeue aListener, Enqueue aSpeaker) throws VMException;
    protected Processor.ForInternalUseOnly getForInternalUseOnly();
    protected class ForInternalUseOnly
    {
        public void setVMWhereThisProcessorIsInstalled(VM aVM);
    }
}
```

A.8.11.3. Methods

A.8.11.3.1. `Processor()`

Constructs a `Processor` that is not installed or running in any `VM`.

A.8.11.3.2. `getVMWhereThisProcessorIsInstalled()`

Returns the `VM` where this `Processor` has been installed; `null`, if not.

A.8.11.3.3. `boolean isThisProcessorInstalled()`

Returns `true` if this `Processor` has been installed; `false` otherwise.

A.8.11.3.4. `void process(Dequeue aListener, Enqueue aSpeaker) throws VMException`

This method will be called from a `VM` object instance when appropriate to perform processing with the incoming data in `aListener` and the produced outgoing data in `aSpeaker`. Any errors caught during processing must be captured and, if necessary, wrapped using the `VMException` exception type.

A.8.11.3.5. `Processor.ForInternalUseOnly getForInternalUseOnly()`

For internal use only. Used for tracking which `VM` the `Processor` instance is installed. A class was created since multiple (class) inheritance is not possible in Java.

A.8.11.3.6. `void ForInternalUseOnly. setVMWhereThisProcessorIsInstalled(VM aVM)`

This method notes that this `Processor` instance has been installed in `aVM`.

A.8.12. Class: `ProcessorForAlgorithm`

A.8.12.1. Overview

A `ProcessorForAlgorithm` separates the queuing mechanism used between the `VM` and its installed `Processor` from the `Algorithm` that specific `Processor` will be running.

A.8.12.2. Definition

```
import preneypaul.msc.libs.os.ds.*;
public class ProcessorForAlgorithm extends Processor
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public ProcessorForAlgorithm(Algorithm anAlgorithm);
    public final synchronized void process(Dequeue aListener, Enqueue aSpeaker);
    public String toString();
}
```

A.8.12.3. Methods

See the superclass for unlisted method descriptions.

A.8.12.3.1. `ProcessorForAlgorithm(Algorithm anAlgorithm)`

Constructs a `Processor` that runs the `Algorithm` defined by `anAlgorithm`.

A.8.12.3.2. `String toString()`

Returns a string for logging purposes.

A.8.13. Class: ProcessorForDispatcher

A.8.13.1. Overview

A ProcessorForDispatcher separates the queuing mechanism used between the VM and its installed Processor from the Dispatcher that specific Processor will be running.

A.8.13.2. Definition

```
import preneypaul.msc.libs.os.ds.*;
public class ProcessorForDispatcher extends Processor
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public ProcessorForDispatcher(Dispatcher aDispatcher);
    public final synchronized void process(Dequeue aListener, Enqueue aSpeaker);
    public String toString();
}
```

A.8.13.3. Methods

See the superclass for unlisted method descriptions.

A.8.13.3.1. ProcessorForDispatcher(Dispatcher aDispatcher)

Constructs a Processor that runs the Dispatcher defined by aDispatcher.

A.8.13.3.2. String toString()

Returns a string for logging purposes.

A.8.14. Class: SynchronousDispatcher

A.8.14.1. Overview

An SynchronousDispatcher synchronously broadcasts objects to a set of "listening" objects. Analogously, if one was talking about computer hardware, this class embodies the act of broadcasting data synchronously across the computer bus to some number of hardware processing units. This class is considered a function object.

A.8.14.2. Definition

```
import preneypaul.msc.libs.dp.*;
public final class SynchronousDispatcher implements Dispatcher
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public SynchronousDispatcher();
    public final synchronized void dispatch(final Object anObject,
        final Iterator aListeners);
    public String toString();
}
```

A.8.14.3. Methods

A.8.14.3.1. `SynchronousDispatcher()`

Constructs an instance of an `SynchronousDispatcher`.

A.8.14.3.2. `void dispatch(final Object anObject, final Iterator aListeners)`

Dispatches `anObject` to the listeners defined in the sequence, `aListeners`. The sequence must be of type `VM`, a class, that is also defined in this package.

A.8.14.3.3. `String toString()`

Returns the string `"SynchronousDispatcher"`. Used for logging purposes.

A.8.15. Class: `VM`

A.8.15.1. Overview

A `VM` defines a *virtual machine* (VM) complete with support for installing (via `Installer`), controlling (via `Controller`) a single processing unit (`Processor`). Since a machine is not practically useful unless it is able to accept input and output data, every `VM` is also an implementation of `Queue`; input data is `Enqueued` and output data is `Dequeued`. While the restriction that only one `Processor` can be installed at a time in a `VM`, one can always create a multiprocessing, distributed, parallel, etc. as required `Processor` since the `Processor` class is abstract. Thus, this `VM` class is generic and is able to support arbitrary machine types. The `VM`

class is a standalone virtual machine. To link together VMs to construct a larger virtual machine one must use the VMBus class defined below.

A.8.15.2. Definition

```
import preneypaul.msc.libs.dp.Iterator;
import preneypaul.msc.libs.os.ds.*;
public class VM implements Queue
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public VM();
    public VM(Queue aListenerQueue, Algorithm anAlgorithm, Queue aSpeakerQueue)
        throws VMException;
    public VM(Algorithm anAlgorithm) throws VMException;
    public VM(Processor aProcessor) throws VMException;
    protected final void doClearProcessingThread();
    protected Controller doCreateController();
    protected Thread doCreateProcessingThread();
    protected Installer doCreateProcessorInstaller();
    public Object dequeue();
    public void enqueue(final Object anObject);
    protected void finalize();
    public final Controller getController();
    public Iterator getInternalVMs();
    protected final Dequeue getListenerQueue();
    public final Processor getProcessor() throws VMException;
    public final Installer getProcessorInstaller();
    protected final Enqueue getSpeakerQueue();
    public boolean isDequeueEmpty();
    public boolean isDequeueFull();
    public boolean isEmpty();
    public boolean isEnqueueEmpty();
    public boolean isEnqueueFull();
    public boolean isFull();
    protected final boolean isProcessingThreadRunning();
    public String toString();
    public boolean tryDequeue(ObjectHolder anObject);
    public boolean tryEnqueue(Object anObject);
}
```

A.8.15.3. Methods

See superclass for unlisted method descriptions.

A.8.15.3.1. VM()

Constructs a VM without a Processor. Two ProducerConsumerMonitors are installed for the Enqueue and Dequeue interfaces of this VM object automatically.

A.8.15.3.2. VM(Queue aListenerQueue, Algorithm anAlgorithm, Queue aSpeakerQueue) throws VMException

Constructs a VM with associated queues, aListenerQueue and aSpeakerQueue, and an Algorithm, anAlgorithm. The VM automatically installs anAlgorithm in a ProcessorForAlgorithm instance. The reader should note that aListenerQueue's Dequeue and aSpeakerQueue's Enqueue respective interfaces are passed into the installed Processor's Dequeue and Enqueue process method arguments respectively. Conversely, aListenerQueue's Enqueue and aSpeakerQueue's Dequeue interfaces form the basis of this VM's Queue interface. If it is not possible to create the request virtual machine, an exception is thrown.

A.8.15.3.3. VM(Algorithm anAlgorithm) throws VMException

Constructs a VM with a ProcessorForAlgorithm Processor using anAlgorithm. Two ProducerConsumerMonitors are installed for the Enqueue and Dequeue interfaces of this VM object automatically.

A.8.15.3.4. VM(Processor aProcessor) throws VMException

Constructs a VM with the given Processor, aProcessor. Two ProducerConsumerMonitors are installed for the Enqueue and Dequeue interfaces of this VM object automatically.

A.8.15.3.5. void doClearProcessingThread()

This method sets the reference to the thread running the processor to null. This method should always be called from a terminating java.lang.Thread created via doCreateProcessingThread(). No other method should ever call this method.

A.8.15.3.6. Controller doCreateController()

Creates an appropriate Controller for this VM. This method should never be directly called. Its purpose is to allow VM-derived classes to override the type of Controller instance that gets

created.

A.8.15.3.7. Thread doCreateProcessingThread()

Creates an appropriate java.lang.Thread instance capable of invoking the installed Processor's process method. This method should never be directly called. Its purpose is to allow VM-derived classes to override the java.lang.Thread creation mechanism.

A.8.15.3.8. Installer doCreateProcessorInstaller()

Creates an appropriate Installer for this VM. This method should never be directly called. Its purpose is to allow VM-derived classes to override the type of Installer instance that gets created.

A.8.15.3.9. void finalize()

Ensures that the VM frees all resources that it is making use of, such as stopping a running Processor, etc.

A.8.15.3.10. Controller getController()

Returns the Controller for this VM's installed Processor.

A.8.15.3.11. Iterator getInternalVMs()

Returns, at the VM implementation's discretion, a sequence of VMs that are installed internally forming a machine within this VM. This may be required for development environments and dynamic queries about internal VMs detail.

A.8.15.3.12. Dequeue getListenerQueue()

Returns the Dequeue that must be passed into a Processor's process method.

A.8.15.3.13. `Processor getProcessor() throws VMException`

Returns the `Processor` that is installed. An exception is thrown if no `Processor` is installed.

A.8.15.3.14. `Installer getProcessorInstaller()`

Returns the `Processor Installer` that this `VM` requires.

A.8.15.3.15. `Enqueue getSpeakerQueue()`

Returns the `Enqueue` that must be passed into a `Processor`'s `process` method.

A.8.15.3.16. `boolean isProcessingThreadRunning()`

Returns `true` if the thread running the `Processor` is running; `false` otherwise. **Note:** This is *not* the same as asking if the `Processor` itself is running, that should be done through the `VM`'s `Controller`. This method should be demoted to `protected` scope.

A.8.15.3.17. `String toString()`

Returns a string for logging purposes.

A.8.16. Class: `VMBus`

A.8.16.1. Overview

A `VMBus` defines a virtual machine suitable for conducting "bus" operations; i.e., it serves as a device design to link together other `VMs` into a network forming, in fact, a larger virtual machine which could be further encapsulated within a `Processor` device inside another `VM` object instance. The `Collector` class is required to capture incoming data that is in some `Dequeue` and place it in an `Enqueue` (of another `VM`).

A.8.16.2. Definition

```
import preneypaul.msc.libs.dp.*;
public class VMBus extends VM
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public VMBus(Dispatcher aDispatcher) throws VMException;
    public Iterator getListeners();
    public synchronized void startListeningTo(VM aVM);
    public void startSpeakingTo(VM aVM);
    public synchronized void stopListeningTo(VM aVM);
    public void stopSpeakingTo(VM aVM);

    protected class Collector
    {
        public Collector(VM aVM);
        public void startCollecting();
        public void stopCollecting();
        protected void finalize();
    }
}
```

A.8.16.3. Methods

A.8.16.3.1. `VMBus(Dispatcher aDispatcher) throws VMException`

Constructs a `VM` with a `ProcessorForDispatcher Processor` using `aDispatcher`. Two `ProducerConsumerMonitors` are installed for the `Enqueue` and `Dequeue` interfaces of this `VM` object automatically.

A.8.16.3.2. `Iterator getListeners()`

Returns a sequence of `VMs` that are listening (i.e., via their `Enqueue` interfaces) for incoming data.

A.8.16.3.3. `void startListeningTo(VM aVM)`

Tells this `VMBus` instance to start listening (via the `Enqueue` interface) to `aVM`'s `Dequeue` interface for data.

A.8.16.3.4. `void startSpeakingTo(VM aVM)`

Tells this `VMBus` instance that it is to start speaking (via the `Dequeue` interface) to `aVM`'s `Enqueue` interface with its output data. This method won't work, however, unless `aVM`'s `startListeningTo`

method is also called as is proper.

A.8.16.3.5. `void stopListeningTo(VM aVM)`

Tells this `VMBus` to stop listening to `aVM`'s `Dequeue` interface for incoming data.

A.8.16.3.6. `void stopSpeakingTo(VM aVM)`

Tells this `VMBus` to stop speaking to `aVM`'s `Enqueue` interface for incoming data. `aVM`'s `stopListeningTo` method should be called whenever this method is called.

A.8.16.3.7. `Collector.Collector(VM aVM)`

Creates a `Collector` that `dequeues` incoming data from `aVM` and `enqueues` it in the `VMBus` object which created the `Collector`.

A.8.16.3.8. `void Collector.startCollecting()`

Starts the daemon `java.lang.Thread` that collects incoming data.

A.8.16.3.9. `void Collector.stopCollecting()`

Stops the daemon `java.lang.Thread` that collects incoming data.

A.8.16.3.10. `void Collector.finalize()`

Forces the daemon `java.lang.Thread` that collects incoming data to stop.

A.8.17. Class: `VMException`

A.8.17.1. Overview

A `VMException` is identical in every respect to its parent class, `OSException`. A new class was created however, to differentiate between exceptions thrown from the `preneypaul.msc.libs.os.vm` package from exceptions thrown elsewhere.

A.8.17.2. Definition

```
import preneypaul.msc.libs.os.OSException;
public class VMException extends OSException
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public VMException();
    public VMException(Exception anException);
    public VMException(String aString);
    public VMException(Throwable aThrowable);
}
```

A.8.17.3. Methods

See corresponding superclass constructors for descriptions.

APPENDIX B.
Demonstration Code

B.1. Overview

This appendix contains all of the `public` and `protected` `classes` and `interfaces` for all of this thesis' kernel code. The code is presented hierarchically first by `package` and then by `class/interface` in alphabetical order. All `public` and `protected` methods have their prototypes given and have their functionality described. All of the code uses valid Java syntax except code listed for Java classes, since the method definition is omitted. However, in all other respects, all Java classes listed herein are syntactically correct.

B.2. Package: `preneypaul.distdemo`

B.2.1. Overview

This package contains the bootstrap code for the example programs used in this thesis.

B.2.2. Classes Hierarchy

```
(java.lang.Object)
    (java.awt.Component)
        (java.awt.Container)
            (com.sun.java.swing.JComponent)
                (com.sun.java.swing.JPanel)
                    AutoscrollListPanel
                    MonitorPanel
                    UserJobControlInterfacePanel
                (java.awt.Panel)
                    (java.applet.Applet)
                        (com.sun.java.swing.JApplet)
                            UserJobControlInterfaceApplet
                (java.awt.Window)
                    (java.awt.Frame)
                        (com.sun.java.swing.JFrame)
                            TaskDesktop
    (com.sun.java.swing.DefaultDesktopManager)
        TaskDesktopManager
```

B.2.3. Classes

```
AutoscrollListPanel
MonitorPanel
TaskDesktop
TaskDesktopManager
```

```
UserJobControlInterfaceApplet
UserJobControlInterfacePanel
```

B.2.4. Interfaces

```
MonitorOutput
VMInfo
```

B.2.5. Class: AutoscrollListPanel

B.2.5.1. Overview

This class extends a JPanel to contains a JList (with a JScrollPane) that adds Strings to the bottom of the list. As items are added, the list is automatically scrolled such that the last item in the list is always visible. This class is used to show the various outputs in the example programs.

B.2.5.2. Definition

```
import java.awt.LayoutManager;
import com.sun.java.swing.*;
public class AutoscrollListPanel extends JPanel
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public AutoscrollListPanel();
    public AutoscrollListPanel(LayoutManager layout);
    public AutoscrollListPanel(LayoutManager layout, boolean isDoubledBuffered);
    public AutoscrollListpanel(boolean isDoubledBuffered);
    public void write(String aString);
}
```

B.2.5.3. Methods

See the superclass for unlisted method descriptions.

B.2.5.3.1. void write(String aString)

This method adds aString to the last line of the list. The scroll bar is automatically set to ensure that aString is showing in the list.

B.2.6. Class: MonitorPanel

B.2.6.1. Overview

A MonitorPanel is a JPanel that has a JTabbedPane control, each of which contain a AutoscollListPanel. MonitorPanels are used to show different types of output, such as debug, timing, and "normal" outputs, for a specific VMInfo instance.

B.2.6.2. Definition

```
import java.awt.LayoutManager;
import com.sun.java.swing.*;
public class MonitorPanel extends JPanel implements MonitorOutput
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public MonitorPanel();
    public MonitorPanel(LayoutManager layout);
    public MonitorPanel(LayoutManager layout, boolean isDoubleBuffered);
    public MonitorPanel(boolean isDoubleBuffered);
    public synchronized void writeToMonitor(String aCategory, String aString);
}
```

B.2.6.3. Methods

See the superclasses for unlisted method descriptions.

B.2.6.3.1. void writeToMonitor(String aCategory, String aString)

Writes to the AutoscrollListPanel's write method aString found on the JTabbedPane pane, aCategory. If no tabbed pane for aCategory exists, then one is automatically created.

B.2.7. Interface: MonitorOutput

B.2.7.1. Overview

A MonitorOutput represents any type of output device that can output Strings differentiated by some category of output. An example of this is the MonitorPanel class.

B.2.7.2. Definition

```
public interface MonitorOutput
{
    static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";
    void writeToMonitor(String aCategory, String aString);
}
```

B.2.7.3. Methods

B.2.7.3.1. `void writeToMonitor(String aCategory, String aString)`

Causes the string, `aString`, to be output to the log represented by `aCategory`.

B.2.8. Class: `TaskDesktop`

B.2.8.1. Overview

A `TaskDesktop` instance is a window, specifically a `JFrame`, that shows a `MonitorPanel` for some `VMInfo` instance. A `VMInfo` instance is, in this thesis, an example program that demonstrates the functionality of this kernel. A `TaskDesktop` window has a menu that allows the user to request that the (i) monitor window be shown if it is not; (ii) more instances of the example program be run; and (iii) to exit from the `TaskDesktop`. Upon startup, the `TaskDesktop` runs a single instance of the example program.

B.2.8.2. Definition

```
import java.awt.event.ActionListener;
import java.util.Hashtable;
import com.sun.java.swing.*;
public class TaskDesktop extends JFrame implements ActionListener, MonitorOutput
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public TaskDesktop();
    public TaskDesktop(VMInfo aVMInfo);
    public void actionPerformed(ActionEvent e);
    public JDesktopPane getTaskDesktopPane();
    public void writeToMonitor(String aCategory, String aString);
}
```

B.2.8.3. Methods

See the superclasses for unlisted method descriptions.

B.2.8.3.1. `TaskDesktop(VMInfo aVMInfo)`

Constructs a `TaskDesktop` for the specified example demonstration program, `aVMInfo`. The `TaskDesktop` superclass' title will be set to `aVMInfo.getTitle()`.

B.2.8.3.2. `void actionPerformed(ActionEvent e)`

Performs the processing of any of the menu items chosen from the menu bar.

B.2.8.3.3. `JDesktopPane getTaskDesktopPane()`

Returns the `JDesktopPane` that is being used within the `TaskDesktop` window.

B.2.8.3.4. `void writeToMonitor(String aCategory, String aString)`

Causes the string, `aString`, to be output to the log represented by `aCategory` for the monitor window associated (and contained) with a specific `TaskDesktop` window.

B.2.9. Class: `TaskDesktopManager`

B.2.9.1. Overview

Each `TaskDesktopManager` instance is responsible for ensuring that `JInternalFrames` do not become resized or moved to outside any of the side boundaries of its parent frame window, a `JFrame`. In this thesis, the parent window is always of type `TaskDesktop`.

B.2.9.2. Definition

```
import com.sun.java.swing.*;
public class TaskDesktopManager extends DefaultDesktopManager
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public TaskDesktopManager();
    public void beginResizingFrame(JComponent aFrame, int aDirection);
    public void endResizingFrame(JComponent aFrame);
```

```
    public void setBoundsForFrame(JComponent aFrame, int x, int y, int w, int h);

    protected static final String RESIZING = "RESIZING";
}
```

B.2.9.3. Methods

See the superclass for unlisted method descriptions.

B.2.10. Class: `UserJobControlInterfaceApplet`

B.2.10.1. Overview

A `UserJobControlInterfaceApplet` is the key `java.applet.Applet`-derived class that a web browser will download in order to run the thesis code examples. The applet's client area contains a `UserJobControlInterfacePanel` which provides all of the functionality that the user sees.

B.2.10.2. Definition

```
import com.sun.java.swing.*;
public class UserJobControlInterfaceApplet extends JApplet
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public UserJobControlInterfaceApplet();
    public String getAppletInfo();
    public void init();
}
```

B.2.10.3. Methods

See the superclass for unlisted method descriptions.

B.2.10.3.1. `String getAppletInfo()`

Returns the thesis code copyright message.

B.2.11. Class: `UserJobControlInterfacePanel`

B.2.11.1. Overview

`UserJobControlInterfacePanel` is a `JPanel`-derived class that provides all of the functionality that the user sees in a `UserJobControlInterfaceApplet` instance. It contains a listing of all of the nodes that the code will run on as well as an interactive listing of all of the example "programs" that can be run across those nodes. A button is provided to run a selected example program.

B.2.11.2. Definition

```
import java.awt.LayoutManager;
import java.awt.event.ActionListener;
import com.sun.java.swing.*;
import com.sun.java.swing.event.ListSelectionListener;

class UserJobControlInterfacePanel extends JPanel
    implements ListSelectionListener, ActionListener
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public UserJobControlInterfacePanel();
    public UserJobControlInterfacePanel(LayoutManager layout,
        boolean isDoubledBuffered);
    public UserJobControlInterfacePanel(boolean isDoubledBuffered);
    public void actionPerformed(ActionEvent e);
    public void invokeTask_ActionPerformed(ActionEvent actionEvent);
    public void populate(String aBrokerAddress);
    public void task_ValueChanged(ListSelectionEvent listSelectionEvent);
    public void valueChanged(ListSelectionEvent e);
}
```

B.2.11.3. Methods

See the superclasses for unlisted method descriptions.

B.2.11.3.1. `void actionPerformed(ActionEvent e)`

Forwards any button events to the `invokeTask_ActionPerformed` method. All other events are ignored.

B.2.11.3.2. `void invokeTask_ActionPerformed(ActionEvent actionEvent)`

For the selected example program, encapsulated in an internally stored `VMInfo` object instance,

this method invokes that `VMInfo` instance's `execute` method.

B.2.11.3.3. `void populate(String aBrokerAddress)`

This method populates all of the available nodes' internet addresses.

B.2.11.3.4. `void task_ValueChanged(ListSelectionEvent listSelectionEvent)`

This method is responsible for ensuring that a description of the example program appears for a selected example program. This allows the user to know what he/she is selecting.

B.2.11.3.5. `void valueChanged(ListSelectionEvent e)`

For any events originating from the list of example programs, this method forwards processing to the `task_ValueChanged` method. All other events are ignored.

B.2.12. Interface: `VMInfo`

B.2.12.1. Overview

The `VMInfo` interface is essentially a 3-tuple of program title, program description, and a method to create and run some task.

B.2.12.2. Definition

```
public interface VMInfo
{
    static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    void execute();
    String getDescription();
    String getTitle();
}
```

B.2.12.3. Methods

B.2.12.3.1. `void execute()`

Creates and runs some task.

B.2.12.3.2. `String getDescription()`

Returns a String describing the task that the execute method creates and runs.

B.2.12.3.3. `String getTitle()`

Returns a String with the title of the task that the execute method creates and runs.

B.3. Package: `preneypaul.distdemo.tasks`

B.3.1. Overview

This package defines classes for the example program tasks that will be demonstrated by `VMInfo`-derived objects.

B.3.2. Classes Hierarchy

```
(java.lang.Object)
    ComputeSumOfRandomIntegerArray_1CPU
    ComputeSumOfRandomIntegerArray_nCPUs
```

B.3.3. Classes

```
ComputeSumOfRandomIntegerArray_1CPU
ComputeSumOfRandomIntegerArray_nCPUs
```

B.3.4. Class: `ComputeSumOfRandomIntegerArray_1CPU`

B.3.4.1. Overview

The `ComputeSumOfRandomIntegerArray_1CPU` represents an example program that computes the sum of a randomly generated integer array for a single CPU. This example program runs on the client's machine only.

B.3.4.2. Definition

```
import preneypaul.distdemo.*;
public class ComputeSumOfRandomIntegerArray_1CPU implements VMInfo
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public ComputeSumOfRandomIntegerArray_1CPU();
```

```
    public void execute();
    public String getDescription();
    public String getTitle();
    public String toString();
}
```

B.3.4.3. Methods

See the superclass for unlisted method descriptions.

B.3.4.3.1. `ComputeSumOfRandomIntegerArray_1CPU()`

This constructor builds the machine that is responsible for computing the sum of a random integer array on the client's machine. Specifically, the machine deployed is best illustrated by the following code fragment:

```
VM theRenderInput = new VM(new Algorithm_RenderInput());
VM theGenerateArray = new VM(new Algorithm_GenerateArray());
VM theSumArray = new VM(new Algorithm_SumArrayOnClient());
VM theRenderOutput = new VM(new Algorithm_RenderOutput());

VMBus theDispatchers = new VMBus[4];
theDispatchers[0] = new VMBus(new SynchronousDispatcher());
theDispatchers[0].startSpeakingTo(theRenderInput);

theDispatchers[1] = new VMBus(new SynchronousDispatcher());
theDispatchers[1].startListeningTo(theRenderInput);
theDispatchers[1].startSpeakingTo(theGenerateArray);

theDispatchers[2] = new VMBus(new SynchronousDispatcher());
theDispatchers[2].startListeningTo(theGenerateArray);
theDispatchers[2].startSpeakingTo(theSumArray);

theDispatchers[3] = new VMBus(new SynchronousDispatcher());
theDispatchers[3].startListeningTo(theSumArray);
theDispatchers[3].startSpeakingTo(theRenderOutput);
```

B.3.4.3.2. `void execute()`

This method enqueues a properly formatted `CallReturnSpace` instance into the machine's start state.

B.3.4.3.3. `String toString`

Returns a string for logging purposes.

B.3.5. Class: ComputeSumOfRandomIntegerArray_2CPUs

B.3.5.1. Overview

The ComputeSumOfRandomIntegerArray_2CPUs represents an example program that computes the sum of a randomly generated integer array across two CPUs. For time reasons, this was never extended to be an arbitrary set of 2 CPUs.

B.3.5.2. Definition

```
import preneypaul.distdemo.*;
public class ComputeSumOfRandomIntegerArray_2CPUs implements VMInfo
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public ComputeSumOfRandomIntegerArray_2CPUs(final String aBrokerHost);
    public static void main(String[] args);
    public void execute();
    public String getDesription();
    public String getTitle();
    public String toString();
}
```

B.3.5.3. Methods

See the superclass for unlisted method descriptions.

B.3.5.3.1. ComputeSumOfRandomIntegerArray_2CPUs(final String aBrokerHost)

This constructor builds the half-machine that is responsible for computing the sum of a random integer array on the client side. Specifically, this half-machine deployed is best illustrated by the following code fragment:

```
VM theRenderInput = new VM(new Algorithm_RenderInput());
VM theGenerateArray = new VM(new Algorithm_GenerateArray());
VM theSumArray = new VM(new Algorithm_SumArrayOnClient());
VM theReceiveResultFromBroker = new VM(new Algorithm_RenderOutput());
VM theTransmitArrayToBroker = new VM(new Algorithm_TransmitArrayToBroker(...));

VMBus theDispatchers = new VMBus[4];
theDispatchers[0] = new VMBus(new SynchronousDispatcher());
theDispatchers[0].startSpeakingTo(theRenderInput);

theDispatchers[1] = new VMBus(new SynchronousDispatcher());
theDispatchers[1].startListeningTo(theRenderInput);
theDispatchers[1].startSpeakingTo(theGenerateArray);
```

```
theDispatchers[2] = new VMBus(new SynchronousDispatcher());
theDispatchers[2].startListeningTo(theGenerateArray);
theDispatchers[2].startSpeakingTo(theSumArray);

theDispatchers[3] = new VMBus(new SynchronousDispatcher());
theDispatchers[3].startListeningTo(theSumArray);
theDispatchers[3].startSpeakingTo(theRenderOutput);
```

Some of the necessary code above has been omitted and can be identified by the use of an ellipsis (...).

B.3.5.3.2. `void main(String[] args)`

This method is responsible for constructing the remote half-machine that performs the actual sum of the array. The machine created is as follows:

```
VM transmitResultToClient = new VM(new Algorithm_TransmitResultToClient());
VM sumArray = new VM(new Algorithm_SumArray());
VM receiveArrayFromClient = new VM(new Algorithm_ReceiveArrayFromClient());

VMBus dispatchers[] = new VMBus[2];
dispatchers[0] = new VMBus(new AsynchronousDispatcher());
dispatchers[0].startListeningTo(receiveArrayFromClient);
dispatchers[0].startSpeakingTo(sumArray);

dispatchers[1] = new VMBus(new AsynchronousDispatcher());
dispatchers[1].startListeningTo(sumArray);
dispatchers[1].startSpeakingTo(transmitResultToClient);
```

B.3.5.3.3. `void execute()`

This method enqueues a properly formatted `CallReturnSpace` instance into the machine's start state.

B.3.5.3.4. `String toString()`

Returns a string for logging purposes.

B.4. Package: `preneypaul.distdemo.tasks.sum`

B.4.1. Overview

This package contains all of the algorithms and any necessary associated classes required for this thesis' example programs.

B.4.2. Classes Hierarchy

```
(java.lang.Object)
    Algorithm_GenerateArray
    Algorithm_ReceiveArrayFromClient
    Algorithm_ReceiveResultFromBroker
    Algorithm_RenderInput
    Algorithm_RenderOutput
    Algorithm_SumArray
    Algorithm_SumArrayOnClient
    Algorithm_TransmitArrayToBroker
    Algorithm_TransmitResultToClient
    (java.awt.AWTEventMulticaster)
        IntegerArrayAlgorithmInputPanelListenerEventMulticaster
    (java.awt.Component)
        (java.awt.Container)
            (com.sun.java.swing.JComponent)
                (com.sun.java.swing.JInternalFrame)
                    IntegerArrayAlgorithmInputFrame
                (com.sun.java.swing.JPanel)
                    IntegerArrayAlgorithmInputPanel
    (java.beans.SimpleBeanInfo)
        IntegerArrayAlgorithmInputPanelBeanInfo
```

B.4.3. Classes

```
Algorithm_GenerateArray
Algorithm_ReceiveArrayFromClient
Algorithm_ReceiveResultFromBroker
Algorithm_RenderInput
Algorithm_RenderOutput
Algorithm_SumArray
Algorithm_SumArrayOnClient
Algorithm_TransmitArrayToBroker
Algorithm_TransmitResultToClient
IntegerArrayAlgorithmInputFrame
IntegerArrayAlgorithmInputPanel
IntegetArrayAlgorithmInputPanelBeanInfo
IntegerArrayAlgorithmInputPanelListenerEventMulticaster
```

B.4.4. Interfaces

```
IntegerArrayAlgorithmInputPanelListener
```

B.4.5. Class: Algorithm_GenerateArray

B.4.5.1. Overview

This algorithm, with the appropriate input, generates an array of random integers as output.

B.4.5.2. Definition

```
import preneypaul.msc.libs.os.ds.*;
public class Algorithm_GenerateArray implements Algorithm
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public Algorithm_GenerateArray();
    public void algorithm(Object anInput, ObjectHolder anOutput);
    public String toString();
}
```

B.4.5.3. Methods

See the superclass for unlisted method descriptions.

B.4.5.3.1. `void algorithm(Object anInput, ObjectHolder anOutput)`

This algorithm, given a random number seed and an array length, generates an array of random integers of that length. To accomplish this, this method accepts a `CallReturnSpace` object in lieu of `anInput` and outputs a new `CallReturnSpace` via `anOutput` as detailed in the following table:

Input (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
`Integer (java.lang)`	`"seed"`	Seed used by the random number generator when generating the array.
`Integer (java.lang)`	`"count"`	Number of randomly generated Integers to be generated in an array for output.
`MonitorOutput`	`"MonitorLog"`	Contains a reference to a class that permits output to monitors.

Output (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
`Vector (java.util)`	`"IntegerArray"`	A generated array of `"count"` `Integer` elements.
`MonitorOutput`	`"MonitorLog"`	Contains a reference to a class that permits output to monitors. Is the same reference to `"MonitorLog"` received as input.

During the actual call, some information is output to a monitor as well.

B.4.5.3.2. `String toString()`

Returns `"GenerateArray"`.

B.4.6. Class: `Algorithm_ReceiveArrayFromClient`

B.4.6.1. Overview

This algorithm, with the appropriate input, reads an array of integers from a client's socket connection. This algorithm serves as an input device driver.

B.4.6.2. Definition

```
import preneypaul.msc.libs.os.ds.*;
import preneypaul.msc.libs.os.vm.*;
public class Algorithm_ReceiveArrayFromClient implements Algorithm
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public Algorithm_ReceiveArrayFromClient();
    public void algorithm(Object anInput, ObjectHolder anOutput);
    public String toString();
}
```

B.4.6.3. Methods

See the superclass for unlisted method descriptions.

B.4.6.3.1. `void algorithm(Object anInput, ObjectHolder anOutput)`

This algorithm, given a `java.net.Socket` returns only information received from that socket,

name a java.lang.Vector of java.lang.Integers and the host and port that accepts replies to the transmitted datum. To accomplish this, this method accepts a CallReturnSpace object in lieu of anInput and outputs a new CallReturnSpace via anOutput as detailed in the following table:

Input (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
Socket (java.net)	"IncomingRequest"	The TCP socket that will receive a Vector of Integers by Java's serialization mechanism.
Output (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
Vector (java.util)	"IntegerArray"	The array of Integer elements received.
String (java.lang)	"ReplyToHost"	A string that identifies the host that will accept replies.
Integer (java.lang)	"ReplyToPort"	An integer that identifies the host's port to send replies back.

During the actual call, some information is output to the standard output console as well.

B.4.6.3.2. String toString()

Returns "ReceiveArrayFromClient".

B.4.7. Class: Algorithm_ReceiveResultFromBroker

B.4.7.1. Overview

This algorithm, with the appropriate input, receives a computed result (i.e., a sum) from a computing system's broker interface. This algorithm serves as an input device driver.

B.4.7.2. Definition

```
import preneypaul.msc.libs.os.ds.*;
import preneypaul.msc.libs.os.vm.*;
public class Algorithm_ReceiveResultFromBroker implements Algorithm
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public Algorithm_ReceiveResultFromBroker();
    public void algorithm(Object anInput, ObjectHolder anOutput);
    public String toString();
}
```

B.4.7.3. Methods

See the superclass for unlisted method descriptions.

B.4.7.3.1. `void algorithm(Object anInput, ObjectHolder anOutput)`

This algorithm receives the result (i.e., a sum) in the form of a `java.lang.BigInteger` object. To accomplish this, this method accepts a `CallReturnSpace` object in lieu of `anInput` and outputs a new `CallReturnSpace` via `anOutput` as detailed in the following table:

Input (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
`Socket` (`java.net`)	`"IncomingRequest"`	The TCP socket that will receive a `Vector` of `Integers` by Java's serialization mechanism.
`MonitorOutput`	`"MonitorLog"`	Contains a reference to a class that permits output to monitors.
Output (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
`BigInteger` (`java.math`)	`"SumOfArray"`	Sum of an array.
`MonitorOutput`	`"MonitorLog"`	Contains a reference to a class that permits output to monitors. Is the same reference to `"MonitorLog"` received as input.

During the actual call, some information is output to a monitor as well.

B.4.7.3.2. `String toString()`

Returns `"ReceiveResultFromBroker"`.

B.4.8. Class: `Algorithm_RenderInput`

B.4.8.1. Overview

This algorithm, with the appropriate input, outputs a random number generator input seed and an integer count. This algorithm serves as an input device driver.

B.4.8.2. Definition

```
import preneypaul.msc.libs.os.ds.*;
import preneypaul.msc.libs.os.vm.*;
public class Algorithm_RenderInput implements Algorithm
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public Algorithm_RenderInput();
    public void algorithm(Object anInput, ObjectHolder anOutput);
    public String toString();
}
```

B.4.8.3. Methods

See the superclass for unlisted method descriptions.

B.4.8.3.1. `void algorithm(Object anInput, ObjectHolder anOutput)`

This algorithm, given an instance of `IntegerArrayAlgorithmInputFrame`, outputs a random number generator input seed and an integer count, if the user clicks on an "OK" button. Otherwise, nothing is output. To accomplish this, this method accepts a `CallReturnSpace` object in lieu of `anInput` and outputs a new `CallReturnSpace` via `anOutput` as detailed in the following table:

Input (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
IntegerArrayAlgorithmInputFrame	"frame"	The window frame that prompts the user for the required random number input seed and integer count.
MonitorOutput	"MonitorLog"	Contains a reference to a class that permits output to monitors.
Output (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
Integer (java.lang)	"seed"	The random number seed as specied by the user.
Integer (java.lang)	"count"	The number of integers as specified by the user.
MonitorOutput	"MonitorLog"	Contains a reference to a class that permits output to monitors. Is the same reference to "MonitorLog" received as input.

During the actual call, some information is output to a monitor as well.

B.4.8.3.2. `String toString()`

Returns "RenderInput".

B.4.9. Class: `Algorithm_RenderOutput`

B.4.9.1. Overview

This algorithm, with the appropriate input, outputs a `java.math.BigInteger` sum to a `MonitorOutput` window. This algorithm serves as an output device driver.

B.4.9.2. Definition

```
import preneypaul.msc.libs.os.ds.*;
import preneypaul.msc.libs.os.vm.*;
public class Algorithm_RenderOutput implements Algorithm
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public Algorithm_RenderOutput();
```

```
    public void algorithm(Object anInput, ObjectHolder anOutput);
    public String toString();
}
```

B.4.9.3. Methods

See the superclass for unlisted method descriptions.

B.4.9.3.1. `void algorithm(Object anInput, ObjectHolder anOutput)`

This algorithm, given an instances of `BigInteger` and `MonitorOutput`, outputs its decimal (i.e., base 10) value to the `MonitorOutput` device. There is no output produced by this algorithm. To accomplish this, this method accepts a `CallReturnSpace` object in lieu of `anInput` and outputs a new `CallReturnSpace` via `anOutput` as detailed in the following table:

Input (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
`BigInteger` (`java.math`)	`"SumOfArray"`	An integer representing a sum.
`MonitorOutput`	`"MonitorLog"`	Contains a reference to a class that permits output to monitors.

During the actual call, some additional information is output to a monitor as well.

B.4.9.3.2. `String toString()`

Returns `"RenderOutput"`.

B.4.10. Class: `Algorithm_SumArray`

B.4.10.1. Overview

This algorithm, with the appropriate input, outputs the sum of an array of integers.

B.4.10.2. Definition

```
import preneypaul.msc.libs.os.ds.*;
import preneypaul.msc.libs.os.vm.*;
public class Algorithm_SumArray implements Algorithm
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public Algorithm_SumArray();
    public void algorithm(Object anInput, ObjectHolder anOutput);
    public String toString();
}
```

B.4.10.3. Methods

See the superclass for unlisted method descriptions.

B.4.10.3.1. `void algorithm(Object anInput, ObjectHolder anOutput)`

This algorithm, given a `Vector` of `Integers`, outputs the sum of those integers. To accomplish this, this method accepts a `CallReturnSpace` object in lieu of `anInput` and outputs a new `CallReturnSpace` via `anOutput` as detailed in the following table:

Input (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
`Vector` (`java.util`)	`"IntegerArray"`	This array is a `Vector` of `java.lang.Integer`.
`String` (`java.lang`)	`"ReplyToHost"`	A string that identifies the host that will accept replies.
`Integer` (`java.lang`)	`"ReplyToPort"`	An integer that identifies the host's port to send replies back.
Output (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
`BigInteger` (`java.math`)	`"SumOfArray"`	The sum of the array passed as input.
`String` (`java.lang`)	`"ReplyToHost"`	A string that identifies the host that will accept replies.
`Integer` (`java.lang`)	`"ReplyToPort"`	An integer that identifies the host's port to send replies back.

During the actual call, some information is output to the standard output console as well.

B.4.10.3.2. `String toString()`

Returns `"SumArray"`.

B.4.11. Class: `Algorithm_SumArrayOnClient`

B.4.11.1. Overview

This algorithm, with the appropriate input, outputs the sum of an array of integers. It is equivalent to `Algorithm_SumArray` except that it writes information to a `MonitorOutput` device instead of standard output.

B.4.11.2. Definition

```
import preneypaul.msc.libs.os.ds.*;
import preneypaul.msc.libs.os.vm.*;
public class Algorithm_SumArrayOnClient implements Algorithm
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public Algorithm_SumArrayOnClient();
    public void algorithm(Object anInput, ObjectHolder anOutput);
    public String toString();
}
```

B.4.11.3. Methods

See the superclass for unlisted method descriptions.

B.4.11.3.1. `void algorithm(Object anInput, ObjectHolder anOutput)`

This algorithm, given a `Vector` of `Integers`, outputs the sum of those integers. To accomplish this, this method accepts a `CallReturnSpace` object in lieu of `anInput` and outputs a new `CallReturnSpace` via `anOutput` as detailed in the following table:

Input (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
`Vector (java.util)`	`"IntegerArray"`	This array is a `Vector` of `java.lang.Integer`.
`MonitorOutput`	`"MonitorLog"`	Contains a reference to a class that permits output to monitors.
Output (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
`BigInteger (java.math)`	`"SumOfArray"`	The sum of the array passed as input.
`MonitorOutput`	`"MonitorLog"`	Contains a reference to a class that permits output to monitors. Is the same reference to `"MonitorLog"` received as input.

During the actual call, some information is output to a monitor as well.

B.4.11.3.2. `String toString()`

Returns `"SumArrayOnClient"`.

B.4.12. Class: `Algorithm_TransmitArrayToBroker`

B.4.12.1. Overview

This algorithm, with the appropriate input, transmits an array of integers to a specied host's port. This algorithm serves as an output device driver.

B.4.12.2. Definition

```
import preneypaul.msc.libs.os.ds.*;
import preneypaul.msc.libs.os.vm.*;
public class Algorithm_TransmitArrayToBroker implements Algorithm
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public Algorithm_TransmitArrayToBroker(String aHost, int aPort, String aReplyHost,
        int aReplyPort);
    public void algorithm(Object anInput, ObjectHolder anOutput);
    public String toString();
}
```

B.4.12.3. Methods

See the superclass for unlisted method descriptions.

B.4.12.3.1. `Algorithm_TransmitArrayToBroker(String aHost, int aPort, String aReplyHost, int aReplyPort)`

Constructs an `Algorithm_TransmitArrayToBroker` object instance. This constructor remembers the transmission target host's DNS name and server port, i.e., `aHost` and `aPort`, and the host DNS name and port, i.e., `aReplyHost` and `aReplyPort`, in where replies to any sent data can be sent.

B.4.12.3.2. `void algorithm(Object anInput, ObjectHolder anOutput)`

This algorithm, given a `Vector` of `Integers`, outputs that information with a host's DNS name and port, to which responses can be sent, to the server as specified during this object's construction. Nothing is output from this algorithm. To accomplish this, this method accepts a `CallReturnSpace` object in lieu of `anInput` and outputs a new `CallReturnSpace` via `anOutput` as detailed in the following table:

Input (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
`Vector (java.util)`	`"IntegerArray"`	This array is a `Vector` of `java.lang.Integer`.
`MonitorOutput`	`"MonitorLog"`	Contains a reference to a class that permits output to monitors.

During the actual call, some information is output to a monitor as well.

B.4.12.3.3. `String toString()`

Returns `"TransmitArrayToBroker"`.

B.4.13. Class: `Algorithm_TransmitResultToClient`

B.4.13.1. Overview

This algorithm, with the appropriate input, transmits a `java.lang.BigInteger` (i.e., the sum of an array of integers) to a specified host's port. This algorithm serves as an output device driver.

B.4.13.2. Definition

```
import preneypaul.msc.libs.os.ds.*;
import preneypaul.msc.libs.os.vm.*;
public class Algorithm_TransmitResultToClient implements Algorithm
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public Algorithm_TransmitResultToClient();
    public void algorithm(Object anInput, ObjectHolder anOutput);
    public String toString();
}
```

B.4.13.3. Methods

See the superclass for unlisted method descriptions.

B.4.13.3.1. `void algorithm(Object anInput, ObjectHolder anOutput)`

This algorithm, given a `BigInteger`, a host DNS name, and port, outputs that integer to the server running on that host's port. Nothing is output from this algorithm. To accomplish this, this method accepts a `CallReturnSpace` object in lieu of `anInput` and outputs a new `CallReturnSpace` via `anOutput` as detailed in the following table:

Input (Type): CallReturnSpace (see page 89)		
(Class) Type	**Identifier**	**Remarks**
`BigInteger` (`java.math`)	`"SumOfArray"`	An integer.
`String` (`java.lang`)	`"ReplyToHost"`	A string that identifies the host that will accept replies.
`Integer` (`java.lang`)	`"ReplyToPort"`	An integer that identifies the host's port to send replies back.

During the actual call, some information is output to the standard output console as well.

B.4.13.3.2. `String toString()`

Returns `"TransmitResultToClient"`.

B.4.14. Class: `IntegerArrayAlgorithmInputFrame`

B.4.14.1. Overview

This class represents the Java Swing window that is used to prompt the user for information

via `IntegerArrayAlgorithmInputPanel`.

B.4.14.2. Definition

```
import com.sun.java.swing.*;
public class IntegerArrayAlgorithmInputFrame extends JInternalFrame
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public IntegerArrayAlgorithmInputFrame();
    public IntegerArrayAlgorithmInputFrame(String title);
    public IntegerArrayAlgorithmInputPanel getInputPanel();
}
```

B.4.14.3. Methods

See the superclass for unlisted method descriptions.

B.4.14.3.1. `IntegerArrayAlgorithmInputPanel getInputPanel()`

This method returns an instance of `IntegerArrayAlgorithmInputPanel`, enabling the client code

to query the necessary information from that.

B.4.15. Class: IntegerArrayAlgorithmInputPanel

B.4.15.1. Overview

This class represents the Java Swing window surface responsible for querying a random number seed and an integer count from a user. This class is automatically inserted when an instance of IntegerArrayAlgorithmInputFrame is created.

B.4.15.2. Definition

```
import java.awt.LayoutManager;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.util.EventObject;
import com.sun.java.swing.*;
import com.sun.java.swing.event.CaretEvent;
import com.sun.java.swing.event.CaretListener;
public class IntegerArrayAlgorithmInputPanel extends JPanel
    implements CaretListener, ActionListener
{
    public static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    public IntegerArrayAlgorithmInputPanel();
    public IntegerArrayAlgorithmInputPanel(LayoutManager layout);
    public IntegerArrayAlgorithmInputPanel(LayoutManager layout,
        boolean isDoubleBuffered);
    public IntegerArrayAlgorithmInputPanel(boolean isDoubleBuffered);
    public void actionPerformed(ActionEvent e);
    public void addIntegerArrayAlgorithmInputPanelListener(
        IntegerArrayAlgorithmInputPanelListener newListener);
    public void caretUpdate(CaretEvent e);
    protected void fireCancelButton(EventObject newEvent);
    protected void fireOkButton(EventObject newEvent);
    public int getNoOfIntegers();
    public int getSeed();
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
    public void keyTyped(KeyEvent e);
    public void noOfIntegersField_CaretUpdate(CaretEvent caretEvent);
    public void noOfIntegersField_KeyTyped(KeyEvent keyEvent);
    public void randomNoSeedField_CaretUpdate(CaretEvent caretEvent);
    public void randomNoSeedField_KeyTyped(KeyEvent keyEvent);
    public void removeIntegerArrayAlgorithmInputPanelListener(
        IntegerArrayAlgorithmInputPanelListener newListener);
}
```

B.4.15.3. Methods

This class' methods have not been defined as they only serve to enable proper Java Swing input handling as required by the demonstration software.

B.4.16. Interface: `IntegerArrayAlgorithmInputPanelListener`

B.4.16.1. Overview

This interface abstractly represents the `EventListener` interface required for proper input handling within `IntegerArrayAlgorithmInputPanel` instances.

B.4.16.2. Definition

```
import java.util.EventListener;
public interface IntegerArrayAlgorithmInputPanelListener extends EventListener
{
    static final String copyright
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    void cancelButton(EventObject newEvent);
    void okButton(EventObject newEvent);
}
```

B.4.16.3. Methods

This class' methods have not been defined as they only serve to enable proper Java Swing input handling as required by the demonstration software.

B.4.17. Class: `IntegerArrayAlgorithmInputPanelListenerEventMulticaster`

B.4.17.1. Overview

This class represents the `AWTEventListener` interface necessary for proper event multicasting as required to handle input within `IntegerArrayAlgorithmInputPanel` instances.

B.4.17.2. Definition

```
import java.awt.AWTEventMulticaster;
import java.util.EventObject;
public class IntegerArrayAlgorithmInputPanelListenerEventMulticaster extends AWTEventMulticaster
implements IntegerArrayAlgorithmInputPanelListener
{
    public static final String copyright
```

```
        = "Copyright (c) 1999 Paul Preney. All Rights Reserved.";

    protected IntegerArrayAlgorithmInputPanelListenerEventMulticaster(
        IntegerArrayAlgorithmInputPanelListener a,
        IntegerArrayAlgorithmInputPanelListener b);
    public static IntegerArrayAlgorithmInputPanelListener add(
        IntegerArrayAlgorithmInputPanelListener a,
        IntegerArrayAlgorithmInputPanelListener b);
    public static IntegerArrayAlgorithmInputPanelListener remove(
        IntegerArrayAlgorithmInputPanelListener a,
        IntegerArrayAlgorithmInputPanelListener b);
    public void cancelButton(EventObject newEvent);
    public void okButton(EventObject newEvent);
}
```

B.4.17.3. Methods

This class' methods have not been defined as they only serve to enable proper Java Swing input handling as required by the demonstration software.

APPENDIX C.
Sample Captured Software Screens

C.1. Screen Capture of the Main Demonstration Applet Screen

The figure below shows the main screen of the demonstration applet outlined in this thesis. As is shown, the applet is downloaded through a client using a web browser capable of running Java v1.1 applets.

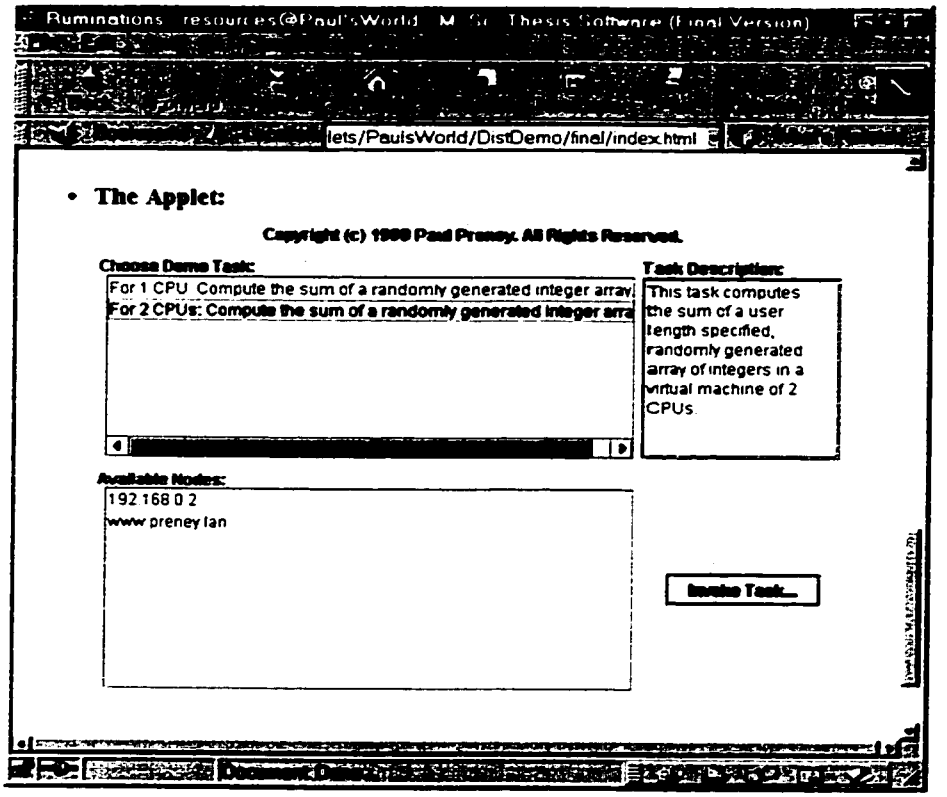

Figure 11. A screen capture of the main applet used to invoke specific example demonstrations.

Based on the IP of the host that delivered the applet and the client's machine a list of nodes appear that allow the software to be run across. Additionally, there are a variety of demonstration tasks that can be chosen to be executed across those nodes by the user.

C.2. A Example Set of Demonstration Screen Captures

Since both the 1-CPU and the 2-CPU demonstrations are nearly identical, only screen captures from a 2-CPU demonstration's job run will be shown here. The figure below shows a screen capture of the window that is shown after selecting the 2-CPU demonstration example. Shown is the user inputting a random number seed and an integer count that will be summed across address boundaries after he/she clicks the "OK" button.

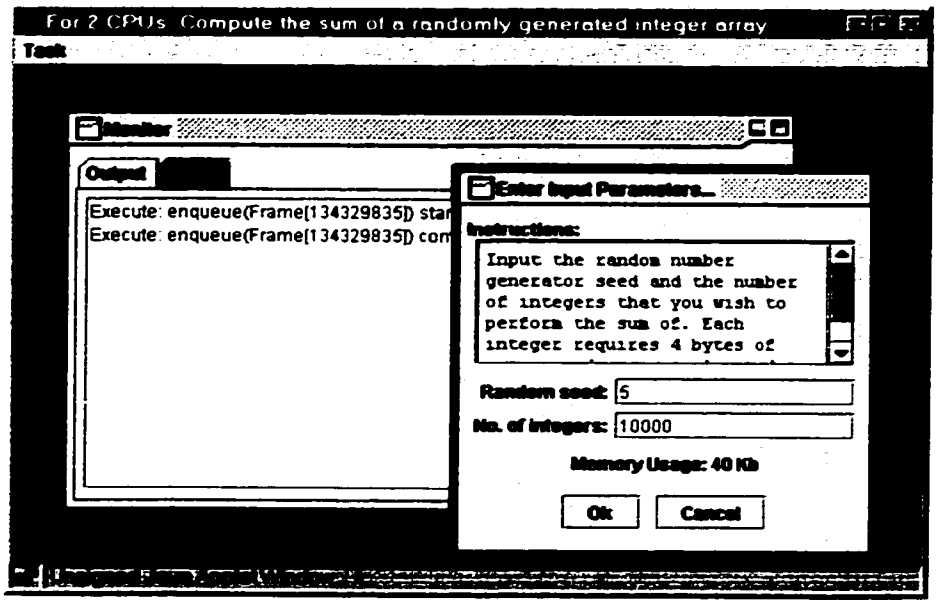

Figure 12. A screen capture of information being inputted in order to start a job run for the 2-CPU demonstration.

Also, shown is the "Monitor" window device. This device serves as an output medium for various types of information reported during the execution of the virtual machine kernel-architecture. The "Monitor" window in Figure 12 is showing its debug output. Clearly visible within the window is the enqueuing of a `java.awt.Frame` into the start state of the 2-CPU demonstration's virtual machine. That `Frame`, entitled "Enter Input Parameters...", appears in the foreground where the user provides the necessary input. In general, all monitor windows have an "Output" tab and dynamically show other tabs when information is output in

specified categories other than "Output".

C.2.1. Server Side Output

The figure below is a screen capture of the standard output as produced on the server side as a result of a 2-CPU job request. This output is equivalent to the "Monitor" window on the client side. It produces output on the standard output device since it the server must be able to run without a graphical user interface present.

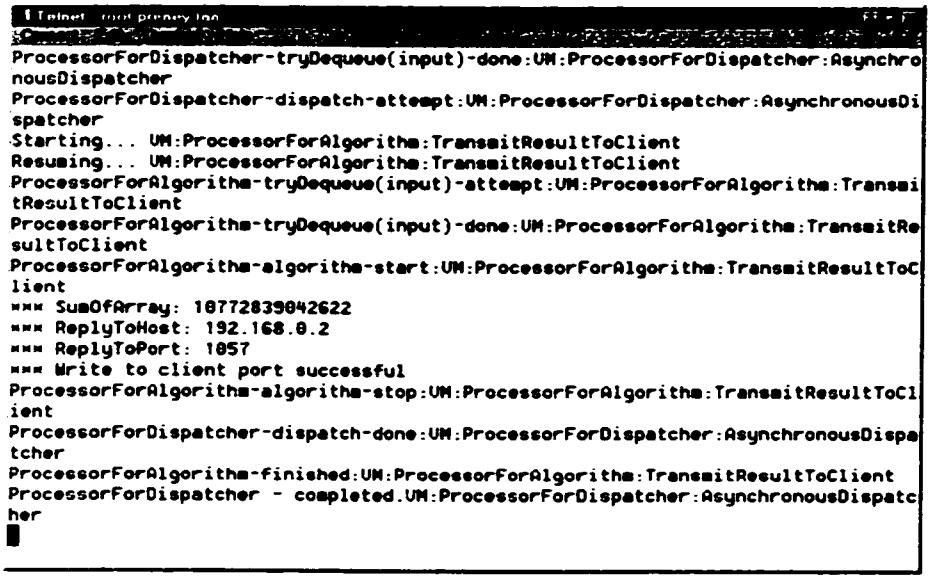

Figure 13. A screen capture of information output on the server side of a 2-CPU demonstration's job run.

The 192.168.0.1 IP address was the address the servers used in this example. As is visible in the figure, the client address was 192.168.0.2. The reader should note that there are client and server components running on each machine. In this appendix, *server* simply refers to the (broker) machine that performs the actual summation.

C.2.2. Client-Side Output

At the end of any computation demonstration in this thesis a sum of an array of integers is

output. The figure below, shows the sum of the job submitted in Figure 12 above.

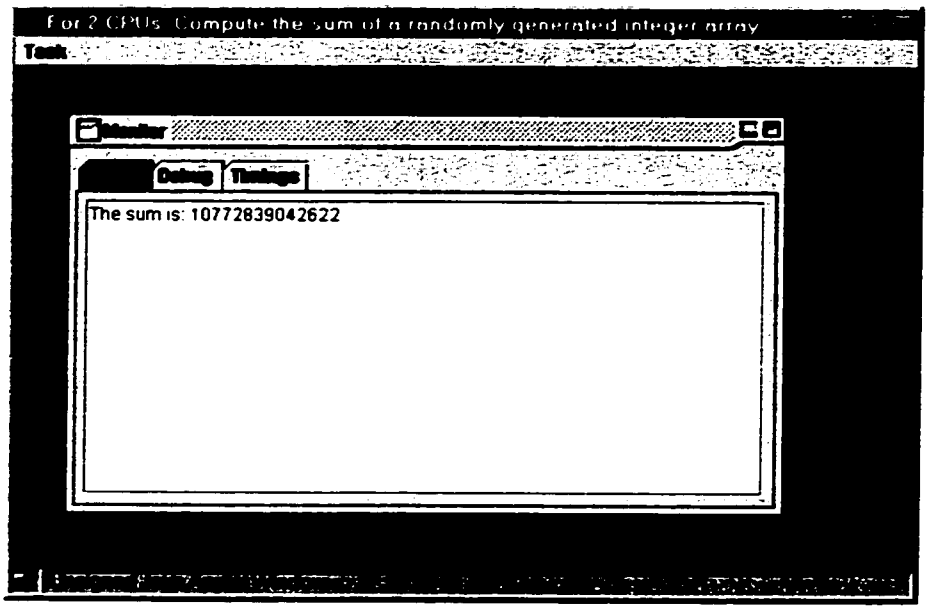

Figure 14. A screen capture of the output on the Monitor device for a 2-CPU demonstration job run.

One should notice that an additional "Timings" tab has appeared since invoking any job causes some timing output to be generated on the monitor device. The output of the "Debug" tab (see Figure below) shows the job enqueued into the virtual machine and the computation of each of its internal (virtual) machine components on the client's machine. The algorithm names correspond to classes starting with `Algorithm_` found in Appendix B.

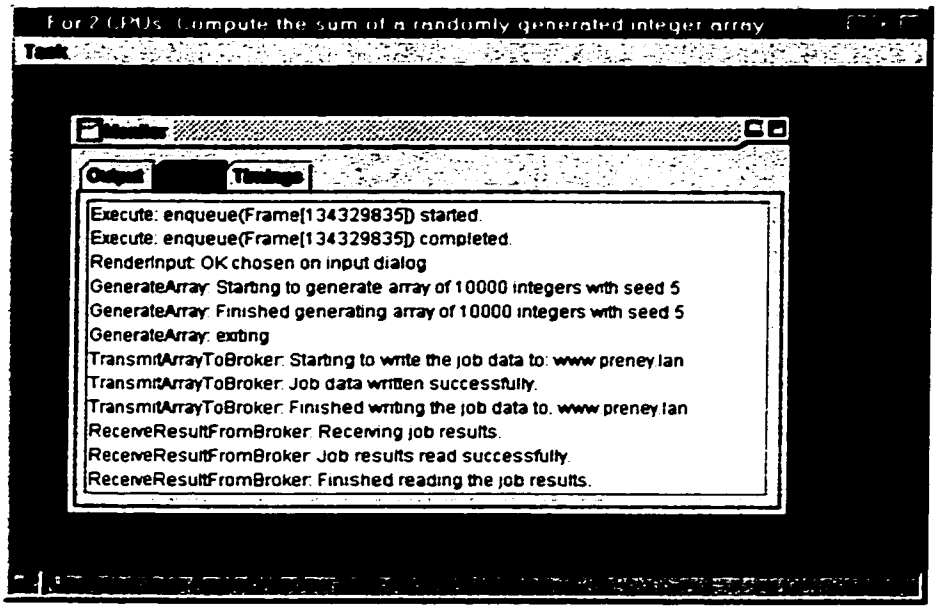

Figure 15. A screen capture of the debug output on the Monitor device for a 2-CPU demonstration job run.

Finally, the various timings regarding the computation time for client-side machine components are shown in the figure below:

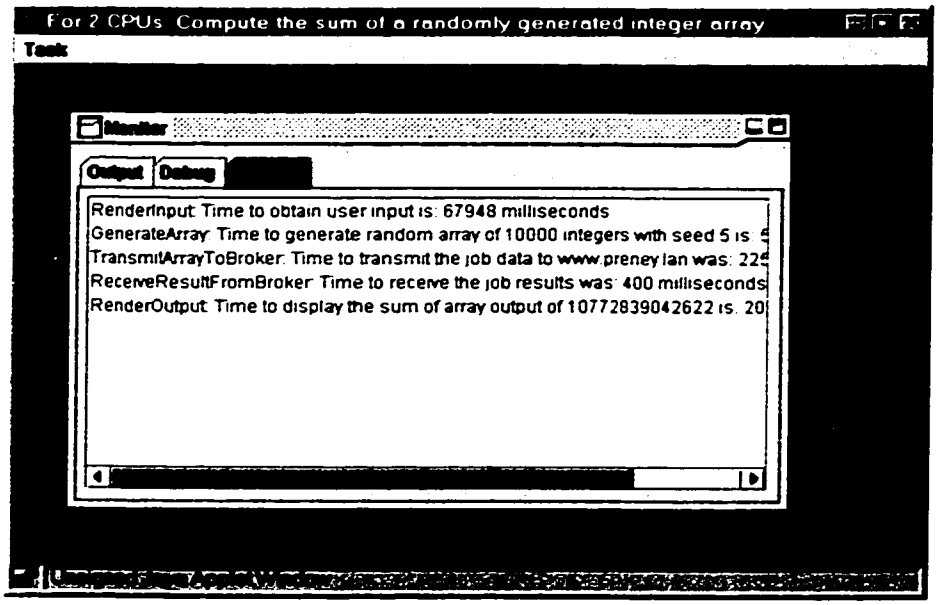

Figure 16. A screen capture of the timing output on the Monitor device for a 2-CPU demonstration job run.

The server used in this example was an Intel 80486 with 16 Mb of RAM running the Linux

2.2.5 kernel with the Apache v1.3.9 web server. This example runs corresponding faster on better machine architectures. Some of the operating systems that the thesis software has been tested under include Windows95/98/NT, OS/2, AIX, IRIX, and Solaris.

References

[AGARW99] Anant Agarwal (1999). "The Oxygen Project: Raw Computation", *Scientific American*, Vol. 281, No. 2, pages 60-63; August 1999.

[AHO86] Alfred V. Aho, Ravi Sethi, & Jeffrey D. Ullman (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts.

[ALMST96] Vicki L. Almstrum, Nell Dale, Anders Berglund, Mary Granger, Joyce Currie Little, Diane M. Miller, Marian Petre, Paul Schragger, & Fred Springsteel (1996). "Evaluation: turning technology from toy to tool, Report of the Working Group on Evaluation", *SIGCSE Bulletin*, Vol. 28, Special Issue, pages 201-217.

[ALPER98] Sherman R. Alpert, Kyle Brown, & Bobby Woolf (1998). *The Design Patterns Smalltalk Companion*. Addison Wesley, Reading, Massachusetts.

[AMES97] Andrea L. Ames, David R. Nadeau, & John L. Moreland (1997). *VRML 2.0 Sourcebook, Second Edition*. John Wiley & Sons, New York, New York.

[ARGON99] Argonne National Laboratory, Mathematics and Computer Science Division (1999). *Grand Challenge Applications*. http://www-fp.mcs.anl.gov/division/research/gca_summary.htm

[ARSIN96] Bogdan G. Arsintescu (1996). "A Method for Analog Circuits Visualization", *Proceedings: IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 454-459; October 7-9, 1996.

[BAKER99a] Mark Baker & Rakjkumar Buyya (1999). "Cluster Computing at a Glance", in: Rakjkumar Buyya, ed., *High Performance Cluster Computing: Programming and Applications, Volume 1*, pages 3-47. Prentice Hall PTR, Upper Saddle River, New Jersey.

[BAKER99b] Mark Baker & Geoffrey Fox (1999). "Metacomputing: Harnessing Informal Supercomputers", in Rakjkumar Buyya, ed., *High Performance Cluster Computing: Programming and Applications, Volume 1*, pages 154-185.

[BENAR90] M. Ben-Ari (1990). *Principles of Concurrent and Distributed Programming*. Prentice Hall, New York, New York.

[BOAS90] Peter van Emde Boas (1990). "Machine Models and Simulations", in: Jan van Leeuwen, ed., *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 1-66. MIT Press, Cambridge, Massachusetts.

[BOSSA96] Pierre-Louis Bossart (1996). "Hypertools in image and volume visualization", *Fourth Annual Tcl/Tk Workshop '96*, pages 221-230; July 10-13, 1996.

[BRONT95] G. H. W. M. Bronts, S. J. Brouwer, C. L. J. Martens, & H. A. Proper (1995). "A Unifying Object Role Modelling Theory", *Information Systems*, Vol. 20, No. 3, pages 213-235.

[BUYYA99a] Rajkumar Buyya (1999). *High Performance Cluster Computing: Architectures and Systems, Volume 1.* Prentice Hall PTR, Upper Saddle River, New Jersey.

[BUYYA99b] Rajkumar Buyya (1999). *High Performance Cluster Computing: Programming and Applications, Volume 2.* Prentice Hall PTR, Upper Saddle River, New Jersey.

[CAMPB90] Neil A. Campbell (1990). *Biology, 2nd edition.* Benjamin/Cummings Publishing Company, Redwood City, California.

[CARZA97] Antonio Carzaniga, Gian Pietra Picco, & Giovanni Vigna. (1997). "Designing Distributed Applications with Mobile Code Paradigms", *Proceedings of the ICSE '97*, pages 22-32. Boston, Massachusetts.

[CASTI96] John L. Casti (1996). *Five Golden Rules: Great Theories of 20th-Century Mathematics —and Why They Matter.* John Wiley & Sons, Inc., New York, New York.

[CALLI95] H. Rebecca Callison (1995). "A Time-Sensitive Object Model for Real-Time Systems", *ACM Transactions on Software Engineering and Methodology*, Vol. 4, No. 3, pages 287-317.

[CHEN76] Peter Pin-Shan Chen (1976). "The Entity-Relationship Model--Toward a Unified View of Data", *ACM Transactions on Database Systems*, Vol. 1, No. 1, pages 9-36; March 1976.

[CHI96] Ed Huai-hsin Chi, John Riedl, Elizabeth Shoop, John V. Carlis, Ernest Retzel, & Phillip Barry (1996). "Flexible Information Visualization of Multivariate Data from Biological Sequence Similarity Searches", in: *Proceedings IEEE Visualization '96*, pages 133-140; October 27-November 1, 1996.

[COPEL99] B. Jack Copeland & Diane Proudfoot (1999). "Alan Turing's Forgotten Ideas in Computer Science", *Scientific American*, Vol. 280, No. 4, pages 98-103; April 1999.

[COULO95] George Coulouris, Jean Dollimore, & Tim Kindberg (1995). *Distributed Systems: Concepts and Design, 2nd edition.* Addison-Wesley, Harlow, England.

[CREAS96] P. N. Creasy & H. A. Proper (1996). "A generic model for 3-dimensional conceptual modelling", *Data & Knowledge Engineering*, Vol. 20, pages 119-161.

[DATE98] C. J. Date & Hugh Darwen (1998). *Foundation for Object/Relational Databases: The Third Manifesto.* Addison-Wesley, Reading, Massachusetts.

[EBERH96] Peter Eberhard & Uwe Neerpasch (1996). "Interactive Modelling of Multibody Systems with an Object Oriented Data Model", *Mathematical Modelling of Systems*, Vol. 2, No. 1, pages 55-68.

[EICK96] Stephen G. Eick & Daniel E. Fyock (1996). "Visualizing corporate data", *IEEE Potentials*, pages 6-11; December '96/January '97.

[ENCAR95] L. M. Encarnação (1995). "Adaptivity in Graphical User Interfaces: An Experimental Framework", *Comput. & Graphics*, Vol. 19, No. 6, pages 873-884.

[FAUST96] Nickolas Faust (1996). "OpenGL VGIS", *Proceedings of the SPIE*, Vol. 2740, pages 42-49.

[FIELD88] Anthony J. Field & Peter G. Harrison (1988). *Functional Programming.* Addison-Wesley, Wokingham, England.

[FLYNN96] Michael J. Flynn & Kevin W. Rudd (1996). "Parallel Architectures", *ACM Computing Surveys*, Vol. 28, No. 1, pages 67-70.

[FORGA96] Adam B. Forgang, Bernd Hamann, & Carl F. Cerco (1996). "Visualization of Water Quality Data for the Chesapeake Bay", in: *Proceedings IEEE Visualization '96*, pages 417-420; October 27-November 1, 1996.

[FORMA94] Ira R. Forman, Scott Danforth, & Hari Madduri (1994). "Composition of Before/After Metaclasses in SOM", *OOPSLA '94 Conference Proceedings*, pages 427-439.

[FORMA99] Ira R. Forman & Scott H. Danforth (1999). *Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming.* Addison-Wesley, Reading, Massachusetts.

[GAMMA95] Erich Gamma, Richard Helm, Ralph Johnson, & John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts.

[GEIST94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, & Vaidy Sunderam (1994). *PVM3 User's Guide and Reference Manual*, September 1994. PVM Web site: http://www.lerc.nasa.gov/WWW/ACCL/pvm.html

[GLOBU99] The Globus Project. http://www.globus.org/

[GRAEF93] Goetz Graefe and Diane L. Davison (1993). "Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Execution", *IEEE Transactions on Software Engineering*, Vol. 19, No. 8, pages 749-764; August 1993.

[GRAEF94] Goetz Graefe (1994). "Volcano—An Extensible and Parallel Query Evaluation System", IEEE Transactions on Knowledge and Data Engineering, Vol. 6, No. 1, pages 120-135; February 1994.

[GRAEF96] Goetz Graefe (1996). "Iterators, Schedulers, and Distributed-memory Parallelism", *Software—Practice and Experience*, Vol. 26, No. 4, pages 427-452; April 1996.

[GROPP98] William Gropp, Ewing Lusk, Nathan Doss, & Anthony Skjellum (1998). *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard.* Message Passing Interface Forum, http://www-unix.mcs.anl.gov/mpi/mpich/.

[HALL96] Steven E. Hall, Mohan K. Ramamurthy, Robert B. Wilhelmson, Joel Plutchak, David Wojtowicz, & Mythili Sridhar (1996). "The Weather Visualizer: A Java tool for interactive learning", *IEEE IGARSS '96*, pages 1498-1500.

[HAMME96] Dieter K. Hammer, L. R. Welch, & O. S. van Roosmalen (1996). "A Taxonomy for Distributed Object-Oriented Real-Time Systems", *ACM OOPS Messenger: Special Issue on Object-Oriented Real-Time Systems*, Vol. 7, No. 1, pages 78-85; January 1996.

[HAROL96] Elliotte Rusty Harold (1996). *Java Network Programming.* O'Reilly & Associates, Sebastopol, California.

[HOARE74] C. A. R. Hoare (1974). "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, Vol. 17, No. 10, pages 549-557; October 1974.

[HOARE78] C. A. R. Hoare (1978). "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21, No. 8, pages 666-677; August 1978.

[HYDE96] R. T. Hyde, P. N. Shaw, & D. E. Jackson (1996). "The Evaluation of Integrated Courseware: Can Interactive Molecular Modelling Help Students Understand Three-Dimensional Chemistry?", *Computers Educ.*, Vol. 26, No. 4, pages 233-239.

[KALAW93] Roy S. Kalawsky (1993). *The Science of Virtual Reality and Virtual Environments.* Addison-Wesley Publishing Company, Wokingham, England.

[LEGIO99] Legion: A Worldwide Virtual Computer. http://legion.virginia.edu/

[LEVIN97] Simon A. Levin, Bryan Grenfell, Alan Hastings, & Alan S. Perelson (1997). "Mathematical and Computational Challenges in Population Biology and Ecosystems Science", *Science*, Vol. 275, pages 334-343; January 17, 1997.

[LEXIC87] Lexicon Publications, Inc. (1987). *The New Lexicon Webster's Dictionary of the English Language.* Lexicon Publications, New York, New York.

[LI93] Xin Li & J. Michael Moshell (1993). "Modeling Soil: Realtime Dynamic Models for Soil Slippage and Manipulation", *Computer Graphics Proceedings: SIGGRAPH '93*, pages 361-376; August 1-6, 1993.

[LI96] P. Peggy Li, William H. Duquette, & David W. Curkendall (1996). "RIVA: A Versatile Parallel Rendering System for Interactive Scientific Visualization", *IEEE Transactions on Visualization and Computer Graphics*, Vol. 2, No. 3, pages 186-201; September 1996.

[MATHE96a] Gary Jason Mathews & Syed S. Towheed (1996). "WWW-based data systems for interactive manipulation of science data", *Computer Networks and ISDN Systems*, Vol. 28, pages 1857-1864.

[MATHE96b] G. J. Mathews (1996). "Visualization Blackboard: Visualising Space Science Data in 3D", in: Lloyd Treinish & Deborah Silver, eds., *IEEE Computer Graphics and Applications*, Vol. 16, No. 6, pages 6-9.

[MAX95] Nelson Max & Roger Crawfis (1995). "Advances in Scientific Visualization", *Proceedings of the SPIE*, Vol. 2410, pages 340-345.

[MEYER97] Jon Meyer & Troy Downing (1997). *Java Virtual Machine.* O'Reilly & Associates, Sebastopol, California.

[MPI95] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* http://www.mpi-forum.org/, June 12, 1995.

[MPI97] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface.* http://www.mpi-forum.org/, July 18, 1997.

[NAPS96] Tom Naps, Joe Bergin, Ken Brodlie, Michael Goldweber, Ricardo Jiménez-Peris, Sami Khuri, Marta Patiño-Martínez, Myles McNally, Susan Rodger, & Judith Wilson (1996). "An overview of visualization: its use and design, Report of the Working Group on Visualization", *SIGCSE Bulletin*, Vol. 28, Special Issue, pages 192-200.

[OAKS98] Scott Oaks (1998). *Java Security*. O'Reilly & Associates, Sebastopol, California.

[ONODE90] Tamiya Onodera & Satoru Kawai (1990). "A Formal Model of Visualization in Computer Graphics Systems", in: G. Goos & J. Hartmanis, eds., *Lecture Notes in Computer Science*, Vol. 421, 100 pages, Springer-Verlag, Berlin, Germany.

[PFLEE98] Shari Lawrence Pfleeger (1998). *Software Engineering: Theory and Practice*. Prentice-Hall, Upper Saddle River, New Jersey.

[RAVEN89] Peter H. Raven & George B. Johnson (1989). *Biology, 2nd edition.* Times Mirror/Mosby College Publishing, St. Louis, Missouri.

[SCHAT97] Bruce R. Schatz (1997). "Information Retrieval in Digital Libraries: Bringing Search to the Net", *Science*, Vol. 275, pages 327-334; January 17, 1997.

[SCHRO96] William J. Schroeder, Kenneth M. Martin, & William E. Lorensen (1996). "The Design and Implementation Of An Object-Oriented Toolkit For 3D Graphics And Visualization", *Proceedings IEEE Visualization '96*, pages 93-100; October 27-November 1.

[SCHRO98] Will Schroeder, Ken Martin, & Bill Lorensen (1998). *The Visualization Toolkit, 2nd edition.* Prentice Hall PTR, Upper Saddle River, New Jersey.

[SHANB97] Vivek K. Shanbhag & K. Gopinath (1997). "A C++ Simulator Generator from Graphical Specifications", *Software—Practice and Experience*, Vol. 27, No. 4, pages 395-423; April 1997.

[SILVA99] Luís Moura e Silva & Rakjkumar Buyya (1999). "Parallel Programming Models and Paradigms", in: Rakjkumar Buyya, ed., *High Performance Cluster Computing: Programming and Applications, Volume 2*, pages 4-27. Prentice Hall PTR, Upper Saddle River, New Jersey.

[SIPSE97] Michael Sipser (1997). *Introduction to the Theory of Computation.* PWS Publishing Company, Boston, Massachusetts.

[STALL92] William Stallings (1992). *Operating Systems.* Macmillan Publishing Company, New York, New York.

[TANEN90] Andrew S. Tanenbaum (1990). *Structured Computer Organization, 3rd edition*. Prentice Hall, Englewood Cliffs, New Jersey.

[TOPCU98a] Haluk Topcuoglu, Salim Hariri, Wojtek Furmanski, Jon Valente, Ilkyeun Ra, Dongmin Kim, Yoonhee Kim, Xue Bing, & Baoqing Ye (1998). "The Software Architecture of a Virtual Distributed Computing Environment", To be published in *The Journal of Networks, Software Tools and Applications (Cluster Computing)*.

[TOPCU98b] Haluk Topcuoglu, Salim Hariri, Dongmin Kim, Yoonhee Kim, Xue Bing, Baoqing Ye, Ilkyeun Ra, & Jon Valente (1998). "The Design and Evaluation of a Virtual Distributed Computing Environment", To be published in *The Journalof Networks, Software Tools and Applications (Cluster Computing)*.

[WEINS97] John N. Weinstein, Timothy G. Myers, Patrick M. O'Connor, Stephen H. Friend, Albert J. Fornace Jr., Kurt W. Kohn, Tito Fojo, Susan E. Bates, Lawrence V. Rubinstein, N. Leigh Anderson, John K. Buolamwini, William W. van Osdol, Anne P. Monks, Dominic A. Scudiero, Edward A. Sausville, Daniel W. Zaharevitz, Barry Bunow, Vellarkad N. Viswanadhan, George S. Johnson, Robert E. Wittes, Kenneth D. Paull, "An Information-Intensive Approach to the Molecular Pharmacology of Cancer", *Science*, Vol. 275, pages 343-349; January 17, 1997.

[WITTE96] Craig M. Wittenbrink, Alex T. Pang, & Suresh K. Lodha (1996). "Glyphs for Visualizing Uncertainty in Vector Fields", *IEEE Transactions on Visualization and Computer Graphics*, Vol. 2, No. 3, pages 266-279; September 1996.

[WOOD96] Jason Wood, Ken Brodlie, & Helen Wright (1996). "Visualization Over The World Wide Web And Its Application To Environmental Data", *Proceedings IEEE Visualization '96 Conference*, pages 81-86; October 27 to November 1, 1996.

Vita Auctoris

Paul David Preney was born in 1972 in Windsor, Ontario, Canada. He graduated from F. J. Brennan High School in 1991 after which he went to the University of Windsor where he obtained an honours B. Sc. (Bachelor of Science) degree in Biological Sciences and Computer Science in 1996. While obtaining his undergraduate degree, Paul was Vice President Administration of the Computer Science Society and co-founded the S.O.C.R. (Student Operated Computing Resources) group, which he served as President of for two years. After obtaining his undergraduate degree, he worked as a consultant in the insurance and automotive industries until he started his M. Sc. (Master's of Science) degree at the University of Windsor. He is currently a candidate for the Master's of Science degree in Computer Science at the University of Windsor and will graduate in the Fall of 1999.